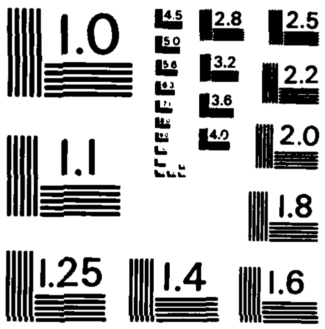

1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

12

RADC-TR-84-27
Final Technical Report
February 1984

AUTOMATING SOFTWARE DESIGN METRICS

The Charles Stark Draper Laboratory, Inc.

Paul A. Szulewski, Nancy M. Sodano, Andrew J. Rosner
and J. B. DeWolf

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

RADC-TR-84-27 has been reviewed and is approved for publication.

APPROVED: Joseph P. Cavano

JOSEPH P. CAVANO
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Acting Technical Director
Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CSDL-R-1662		5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-84-27		
6a. NAME OF PERFORMING ORGANIZATION The Charles Stark Draper Laboratory, Inc.	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)		
6c. ADDRESS (City, State and ZIP Code) 555 Technology Square Cambridge MA 02139		7b. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL (If applicable) COEE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-C-0130		
8c. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441		10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 62702F	PROJECT NO. 5581	TASK NO. 20 / WORK UNIT NO. 42

11. TITLE (Include Security Classification)
AUTOMATING SOFTWARE DESIGN METRICS

12. PERSONAL AUTHOR(S)
Paul A. Szulewski, Nancy M. Sodano, Andrew J. Rosner, J. B. DeWolf

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Sep82 TO Sep83	14. DATE OF REPORT (Yr., Mo., Day) February 1984	15. PAGE COUNT 164

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Software Design Metrics Software Quality Measurements Software Science Automated Design Tools
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Rome Air Development Center has developed the Software Quality Framework as a means to specify software quality goals and measure software quality. Much of the work to date has focused on metrics applicable to software code. This report describes an effort undertaken to measure the quality of software products earlier in the software development life cycle, during the design phase, and to automate the capture of metric data from design media.

Metrics of software quality, primarily those related to the criterion simplicity (or conversely, complexity), were reviewed. This review includes those metrics previously developed in the Software Quality Framework. Two metrics, Halstead's Software Science and McCabe's Cyclomatic Complexity were chosen for their amenability to measurement during design and their potential for automation. Two design media were used: Design Aids for Real-Time Systems (DARTS), an experimental automated design tool developed at the Charles Stark Draper Laboratory; and Ada as a program design language (PDL).

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph P. Cavano	22b. TELEPHONE NUMBER (Include Area Code) (315) 330-7834	22c. OFFICE SYMBOL RADC/COEE

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Automatic measurement of the Halstead and McCabe metrics was implemented as an analysis capability of DARTS. Software designs were encoded into DARTS and subsequently measured for quality and other measurable parameters. These experiments provide some evidence that early measurement can supply both static quality assessment and project planning data which is useful information for managers and designers alike. The Halstead metric was also manually applied to a textbook design represented in an Ada PDL. This experiment showed that it is feasible to use Ada as a design medium, that Halstead metric data can be captured from an Ada Design, and that if an automated Ada PDL is used, there is potential for automated measurement.

Finally, a methodology was proposed for using design metrics in support of an integrated software development environment. This methodology was shown to be capable of providing early measures of software quality, and other planning estimates like delivered source instructions, costs, and schedules.

[1] Ada is a registered trademark of the US Department of Defense (AJPO).

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1.0 INTRODUCTION

People concerned with the evaluation of software products are acutely aware of the need for automated support tools and methods. If software quality could be objectively and automatically assessed early in the life-cycle, provisions could be taken to assure that quality goals are being met. This could ultimately reduce life-cycle costs and result in a more reliable and maintainable product.

Prior work sponsored by the Rome Air Development Center (RADC), has developed the Software Quality Measurement Framework, a means to specify quality goals and measure software quality. This effort enhances that framework by identifying metrics that can be used on software designs, automating these design metrics, and providing a methodology for using metrics, embedded in automated design tools, during the early phases of the software development life-cycle. An experimental design tool, Design Aids for Real-Time Systems (DARTS), is used to illustrate the metrics and methodology. In addition, design metrics in the Ada[2] context are also considered.

1.1 REPORT ORGANIZATION

This report is organized as follows. Section 1 provides background, definitions, and an overview of the research program. Section 2 includes detailed technical data and research results related to the identification and development of automated design metrics. Section 3 describes a methodology developed for using automated design-aid tools and metrics in support of an integrated software development environment. Section 4 summarizes the conclusions of this effort and Section 5 provides a list of recommendations for future research.

The appendices supply A) a list of acronyms, B) DARTS trees for the example designs, C) DARTS design trees of the Halstead metric implementation, D) design trees of the McCabe metric, and E) DARTS data-flow tables for the Experiment Controller example. A list of references and a bibliography of sources used are also included.

[2] Ada is a registered trademark of the U. S. Department of Defense (AJPO).

1.2 HISTORICAL PERSPECTIVE

High quality software is of interest to both the software engineering community and its users. As evidenced by the Software Initiative [DoD82a], recently renamed Software Technology for Adaptable and Reliable Systems (STARS) [DoD 83], which by charter will develop tools and methods to increase the quality of DoD software, software quality will no longer be tested-in, but rather be required and designed-in. An important part of the Initiative is the development of metrics to measure the quality of both the software development process and software products. With some advantageous foresight into this problem, the RADC has been sponsoring research in this technical area, in particular the development of the Software Quality Framework [McC 77] which identifies both user- and management-oriented techniques for quantifying software product quality.

1.2.1 The Software Quality Framework

The initial Software Quality Framework effort, sponsored by RADC and Electronic Systems Division (ESD) under contract F30602-76-C-0147, addressed two major issues, software quality specification and measurement. This effort identified 11 factors in a hierarchical framework for acquisition managers to use to specify, predict, and control software quality. The following definitions are provided.

Software: the programs and documentation associated with and resulting from the software development process.

Quality: a general term applicable to any trait or characteristic, whether individual or generic; a distinguishing attribute which indicates a degree of excellence or identifies the basic nature of something.

Factor: a condition or characteristic which actively contributes to the quality of the software. The following rules apply to the set of software quality factors:

A condition or characteristic which contributes to software quality.

A user-related characteristic.

A relative characteristic between software products.

Criteria: attributes of the software or software-production process by which the factor can be judged and defined. The following rules apply to the criteria:

Attributes of the software or software products of the development process; i.e., criteria are software-oriented while factors are user-oriented.

May display a hierarchical relationship with subcriteria.

May affect more than one factor.

Metrics: quantitative measures of the software attributes related to the quality factors. The measures may be objective or subjective.

The relationship of factors, criteria, and metrics is illustrated in Figure 1. McCall's framework identifies 11 prime factors (correctness, efficiency, integrity, usability, testability, flexibility, reusability, maintainability, reliability, portability, and interoperability) which correspond to user-oriented attributes. Corresponding criteria were established as software-product-oriented attributes. The criteria have a fourfold purpose:

1. To refine the factor.

2. To help describe relationships between factors.

3. To establish a one-to-one relationship between criteria and metrics.

4. To create a natural hierarchy in the framework for factors in software quality.

In 1978, RADC and the U.S. Army Computer Systems Command continued this work with a Metrics Enhancement Study under contract number F30602-78-C-0216. The results of the study, reported in [McC 79], refined the results of the initial study and produced a measurement manual for acquisition managers describing how to apply the framework in the acquisition process.

In 1979, another contract, number F30602-79-C-0267, was awarded to develop an Automated Measurement Tool (AMT). The AMT, delivered to the Air Force in September of 1981, automates the collection of specific metric data from programs written in COBOL, and provides a quality metric assessment.

In 1980, RADC sponsored additional refinements to the framework under contract F30602-80-C-0265 to formulate and validate metrics for interoperability and reusability. This effort resulted in a slightly rearranged framework [Boe 83b] and a new measurement manual [Boe 83a] for acquisition managers.

In 1982, RADC sponsored this effort, under contract F30602-82-C-0130, to improve the framework by identifying metrics useful in the early phases of the software development life-cycle. This was motivated by evidence that it is easier and more cost-efficient to correct software at the requirements and

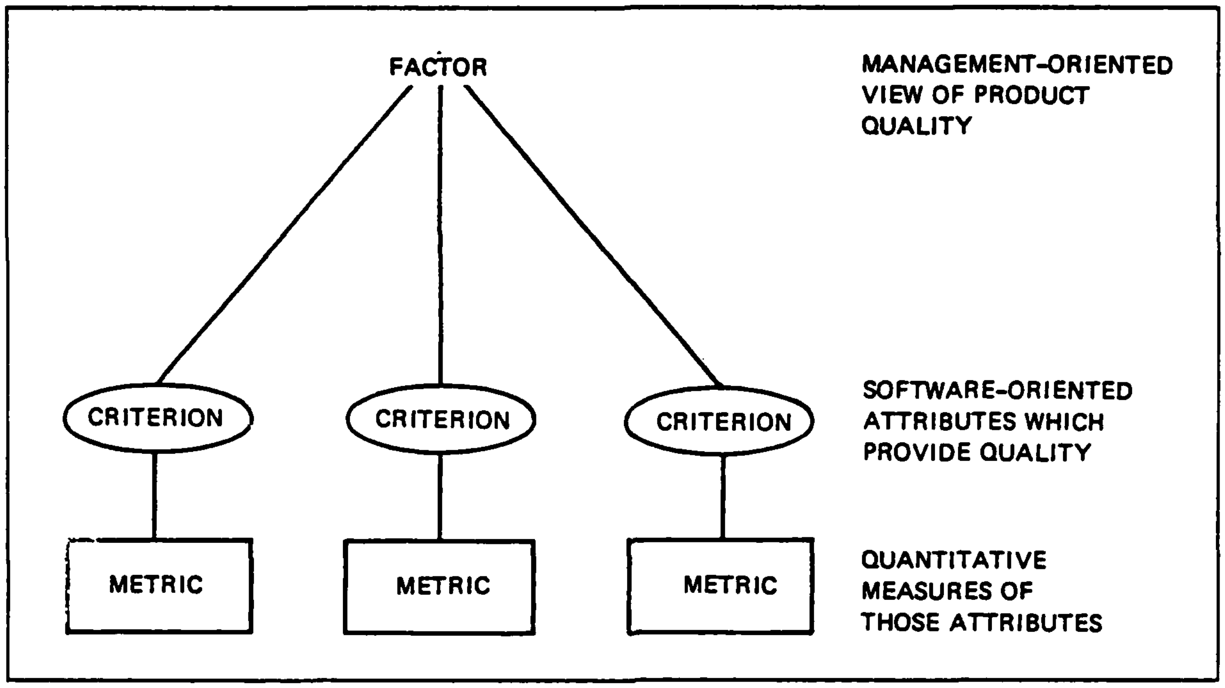

FACTOR
MANAGEMENT-ORIENTED VIEW OF PRODUCT QUALITY
CRITERION
CRITERION
CRITERION
SOFTWARE-ORIENTED ATTRIBUTES WHICH PROVIDE QUALITY
METRIC
METRIC
METRIC
QUANTITATIVE MEASURES OF THOSE ATTRIBUTES

Figure 1. The Software Quality Framework

design phases. Where most of the metrics previously developed were oriented toward manual collection of data from software code, this approach emphasizes automated software design tools for data collection from encoded software design media.

Design tools for the DoD will in the future be increasingly focused on the Ada language and Ada Programming Support Environments (APSEs). Some Ada-specific design tools already exist, such as PDL/Ada [Weg 82] and Byron [Gor 83]. Other design-aid tools will become available as part of the DoD STARS program. Part of this effort has investigated the evaluation of Ada designs with a design metric.

1.2.2 Metrics of Software Quality

The collection of metric data during software design can provide early visibility of the quality of tne developing software product, and better estimates of its size and complexity.

Software quality is that collection of traits or characteristics which imply a degree of excellence or goodness of a software product. This research builds on the contributions of many other software engineering efforts, most notably [McC 77] and [Boe 83b], which have defined and refined a framework for quantifying software quality.

According to the groundrules set in the framework, software quality is measured by the absence, presence, or degree of some identifiable software product attribute. A premise upon which this research is based requires software designs to be viewed as viable and measurable software products, and the criteria, which can be captured from software design media, must be defined. Design metrics should not be dependent on the design medium used but the medium must have the necessary information content. The information necessary for each depends on the metric chosen. The capture of this information can be automated through the use of design-aid tools. Automation improves both the efficiency of the process and the consistency of the data gathered.

Automated design evaluation would be useful to designers, program managers, and program office personnel alike. Designers would be able to quickly compare competing designs and objectively choose the best one. Managers could more easily track the software's development and estimate more accurately its eventual size and completion date. Similarly, program office personnel would have more visibility into the status and quality of the product for which they are responsible.

A more detailed description of the tools and techniques necessary to use design metrics effectively is included as Section 3 of this report.

1.2.3 Automated Software Design Tools

Software at the design phase exists in various product forms, most commonly as flowcharts, program design languages [Cai 75], or other design media (e.g.,[CSDL80] and [Tei 77]). This product is an abstraction of the eventual code, and as such, can be evaluated as a predictor of the quality of the software code end-product. Early indicators of quality are desirable and have the potential to increase reliability and decrease costs. In a subsequent section, this topic is considered with respect to automated design media which, in the authors' opinion, have many advantages over classical design media (e.g., flowcharts).

1.2.3.1 Software Design Media

A software design medium is any form of notation (textual or graphical) for representing and communicating software designs. Design media aid software development personnel by assisting with the following functions [Szu 80].

1. Representation of the system and software architecture at various stages of development.

2. Enforcement of the use of design standards.

3. Implementation planning.

4. Performance and quality assessment.

5. Management visibility and control during implementation.

Most currently available design media are deficient in one or more of the above areas and few are automated. In the following sections a brief survey of current automated design media is included. This survey is not complete, but does illustrate the available automated design media technology.

1.2.3.2 Automated Program Design Languages

Various automated program design languages (PDLs) are available. These structured English processors produce various output to replace traditional flowcharts. These media are easier to read and modify, and can be easily adjusted for any desired level of detail. Some current examples follow.

The Caine, Farber and Gordon PDL [Cai 77] is a tool to aid in designing and documenting a program or system of programs. A design in PDL is written in structured English then submitted to the PDL processor with control information to form procedures. The output is a working design document consisting of a detailed table of contents, a listing of formatted procedures, a call tree, and a cross reference of the procedure calls. This tool evolved from Caine and Gordon's earlier work [Cai 75].

PDL/Ada [Weg 82] was developed for use with the Ada language. It uses a proper subset of the Ada language.

Byron [Gor 83] was developed by Intermetrics specifically for use with the Ada language, although its author claims that its use is not restricted to Ada. Byron is based on Ada such that any legitimate Ada program is also a legitimate Byron specification. It differs from Ada in two respects. Byron allows additional information about a declaration to be associated with it. Second, Byron tools can produce useful output from incomplete specifications. These advantages over pure Ada are discussed in more detail in Section 2.5.

1.2.3.3 Automated Requirements and Design Languages

The Problem Statement Language/ Problem Statement Analyzer (PSL/PSA) [Tei 77], was developed to improve the process of preparing and analyzing software specifications. This automated tool provides a medium which uses objects, relationships between objects, and properties of objects to specify software/system processes. PSA checks consistency of the database and provides a variety of reports.

1.2.3.4 Design Aids for Real-Time Systems (DARTS)

Design Aids for Real-Time Systems (DARTS) [Fur 81] [CSDL82] is a tool developed at The Charles Stark Draper Laboratory, Inc. (CSDL) that assists in defining embedded computer systems through tree-structured graphics, documentation support, and various analysis features. These analysis features provide both static and dynamic software design feedback which can potentially aid in the production of efficient, reliable, and maintainable software systems.

DARTS uses a mix of hierarchy, control and communications primitives, and data structures to represent real-time systems. Requirements are expressed as a functional hierarchy and designs as a tree-structured hierarchy of communicating processes.

Although developed to represent real-time interactions, DARTS can be used for both real-time and non-real-time systems. Through a friendly, menu-oriented interface, a user can build an encoded representation of a system; perform data-flow checking; generate simulations of the design to estimate response time, throughput, and utilization; create a variety of data tables and graphical tree-structured output in a variety of sizes; and, most recently, request an automated quality assessment of a software design.

DARTS has been used throughout this research project as a documentation aid and development test-bed for the software design quality metrics described in the next section. Throughout this report, DARTS trees and tables appear as illustrations.

1.3 PROJECT OVERVIEW

This research project is part of RADC's Software Quality Measurement Program. It has demonstrated that automatic and objective measures of software design quality can be captured from encoded software design media. In addition, a methodology has been provided, useful for designers and program office personnel alike, for incorporating automatic design quality assessment tools into the acquisition and development of future software products.

Various metrics have shown utility as indicators of design quality, but few have been automated and integrated into a design-aid tool. Two metrics were chosen for implementation and integration into an available design-aid tool. These metrics, though added to the DARTS analysis capability, are not constrained to this particular tool. To demonstrate this, one metric is shown to be applicable to Ada designs.

1.3.1 Research Objectives

The objectives of this research program were to

1. develop and validate software design quality metrics, and
2. develop a methodology, consistent with the RADC Software Quality Framework, for
   a. evaluating competitive software designs,
   b. estimating software project planning parameters,
   c. monitoring software product quality.

1.3.2 Technical Approach

To accomplish these objectives, the tasks listed in the following section were defined. These tasks are described as stated in the contractual statement of work, and a summary statement for each task follows the description. Detailed technical data corresponding to these tasks are referenced and follow in the remainder of this report.

- Task 4.1.1 Develop Design Phase Software Quality Metrics

  In this task, design phase-oriented metrics shall be developed and demonstrated to enhance the Software Quality Framework. These metrics shall include, but not be limited to, software science metrics.

  In this task, a survey of the literature was performed, the detailed results of which are described in Section 2.2. The sources are included in the bibliography. Design oriented metrics were examined and shown to be compatible with the RADC Software Quality Framework. The Framework was also reexamined in the light of dividing the total design effort into architectural and detailed design phases. These results are documented in Table 5.

- Task 4.1.2 Automate Design Quality Metrics

  In this task, design quality metrics which are suitable for automation shall be identified. In addition, at least two metrics shall be automated using the DARTS tool. Those metrics not suitable for automation shall also be identified and manual procedures for collecting the data shall be documented.

In this task, metrics suitable for automation were identified and those that were not were listed. These results are summarized in Table 5 and Table 6. Two metrics were chosen as candidates for automation using DARTS. The Halstead metric technique was implemented after requirements and design details were resolved. The McCabe metric was also implemented. Implementation independent details are included in Section 2.2. Requirements and Design information for the DARTS implementation of both metrics is included in Section 2.3.

- Task 4.1.2.1 Apply Metrics To Software Designs

In this task, the metrics shall be applied to a set of designs for which actual code exists.

The literature search turned up many examples of code to use for this validation stage. In some instances, code representing both good and bad coding form was available for functionally equivalent units. These units were translated into a DARTS design and then evaluated by both the Halstead and McCabe metrics. The DARTS trees for these designs appear in Appendix B and the metric analysis is discussed in Section 2.4.

In order to extend this technique to other design media, the Halstead metrics were applied to Ada designs. The results of this task are presented in Section 2.5.

- Task 4.1.2.2 Verify Metrics

In this task, the utility of the metrics as estimators of quality and size (where appropriate) shall be verified.

Empirical data has been generated for a small sample of problems. These results are compared with subjective assessments, and previously published data. The results of this task are summarized in Section 2.4.

- Task 4.1.2.3 Calibrate New DARTS Metrics

In this task, the metrics under development shall be calibrated, if necessary, to improve the measurement technique.

A simple calibration was performed on the Halstead metric. Using the results of Task 4.1.2.1, the design was modified to generate results consistent with those found in the literature. No calibration of the McCabe metric was performed.

- Task 4.1.4 Develop a Methodology for Using Design Metrics

    In this task, a methodology shall be developed to address the following issues.

    1. evaluate competing designs

    2. estimate various project planning parameters

    3. monitor the quality of a software project

    This methodology shall be consistent with the objectives of the Software Quality Framework.

    The results of this task are the topic of Section 3. In that section, a method is described for using Halstead's metrics in the early design phase to estimate the size of an implementation and hence the project cost and schedule.

2.0 ON THE DEVELOPMENT, USE, AND AUTOMATION OF DESIGN METRICS

Software metrics can be useful within the context of an integrated software engineering environment. The purpose of this section is to describe the characteristics of such an environment.

2.1 A CONTEXT FOR METRICS IN THE DEVELOPMENT PROCESS

The software development process consists of personnel engaged in software engineering for the production of software products. Boehm [Boe 81, pp.16] defines software engineering as

"... the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation."

This definition implies skill, innovation, and intuition on the part of the software engineer, as well as the support tools that the engineer uses. Modern programming methods like top-down, stepwise decomposition, information hiding, and modular programming have improved programming style and automated tools have lessened the burden of documentation, program generation, configuration management, and analysis. Yet even today, few tool packages exist that are integrated into a portable product which is useful throughout the development process.

The term "integrated" is stressed to distinguish it from "bundled". Software tools have traditionally evolved from the bundling concept. That is, tools that function well individually have been used together with other similar tools. Data interfaces between tools, when they exist, are set up to provide a continuum of data flow and information from one tool to the next. Often, error-prone human users are these interfaces, and the success of this translation is dependent on the skill of the user. Information can be lost, errors introduced, and product quality suffers. Manpower used in this operation is often wasted. By providing integrated tools, which by design directly interface to each other, the chances for a better quality product are increased and manpower is reduced. Providing such a toolset is one goal of the STARS program. This research supports that goal.

The basic elements of an integrated software engineering environment are skilled personnel, tools, and management and business practices which contribute to the development and maintenance of software products. This environment provides the context for all of the activities associated with software product development during its life-cycle. The life-cycle of a software product begins at the conception of a required capability and ends with the software's eventual retirement from use. In many DoD applications this life-cycle is easily fifteen to twenty years long.

In the sections to follow, a software life-cycle model is described which depends on automated support tools, including software metrics. This model is presented here to define a frame of reference for the succeeding discussion.

2.1.1 A Software Development Process Model

Although this research emphasizes the design phase of a software product, it is important to characterize the total picture to see where it fits in. In this section, a software development process model is described. This model takes into account both the software products (i.e., documentation, code, support tools, etc.) and the management activities (i.e., planning, organizing, staffing, controlling, and assessing). Coordination of engineering and management talent creates quality software. Integrated software tools support both of these disciplines over the life-cycle of the product. The scope of this research extends to both of these domains. Quality metrics can provide an assessment of the quality of the software product, and indirectly, the quality of the process. These issues will be discussed further in succeeding sections.

2.1.2 The Software Development Life Cycle

The software development life-cycle can be simply divided into 6 phases.

- Software Requirements Specification
- Architectural Design (Top Level or Preliminary)
- Detailed Design
- Code and Unit Test
- Integration and Acceptance Test
- Operations and Maintenance

It is often difficult to determine where one phase begins or ends so the following guidelines are offered to precisely define the start and end points of each phase, the corresponding software products, and management activities. This model is generally applicable to medium (i.e., 1 to 4 man-years) or large scale (i.e., greater than 4 man-years) development efforts.

1. Start Requirements Specification Phase

This phase assumes that a prior system specification activity has occurred to determine an appropriate division of basic requirements made between hardware and software. This requires a definition of the system architecture, man-machine interface, and quality goals, and a plan of basic milestones, activities, and schedules. The input to this first software development phase is generally a verbal statement of the user's software needs. This activity should clearly define what the software is to accomplish.

- Activities - Planning, requirements formalization and consistency checking.

- Outputs - Preliminary plans, requirements document.

2. End Requirements Phase, Begin Architectural Design Phase

This endpoint is often reached by a formal requirements review and acceptance activity. The Architectural Design Phase, often refered to as Top Level or Preliminary, should produce a functional architecture which is sufficiently detailed in function, performance and interface definitions, to allow both users and designers to be confident that requirements can be met and the design implemented. Development data, sufficient for product and process metrics, may be collected on available documentation. Support tools, necessary for the effort, should be acquired or developed.

- Activities - Architectural design, planning.

- Outputs - Detailed development plans, architectural design document.

3. End Architectural Design Phase, Begin Detailed Design Phase

This endpoint is often reached by a Preliminary Design Review activity. It may be formal or informal.

- Activities - Software component architecture definition, module design, data base design.

- Outputs - Detailed design specifications for each module, data base specifications, preliminary test and integration plans, draft user's manual, product measures, and process measures.

4. End Detailed Design Phase, Start Code/Unit Test Phase

This endpoint is reached by a Critical Design Review activity. The review, or walkthrough, can be done at the module or program level.

- Activities - Develop code and test each module according to standards set in development plan. Verify completeness and provide requirements traceability. Configuration manage modules. Complete test, acceptance and integration plans, and user's manual.

- Outputs - Verified and tested modules, approved acceptance plan, product measures, and process measures.

5. End Code/Unit Test Phase, Start Integration and Acceptance Test Phase

   This phase ends when all modules have satisfied the unit test requirements.

   - Activities - integration of modules into programs, hardware/software integration and system test. Update detailed design documents to reflect the as-coded version. Provide problem reporting and resolution.

   - Outputs - Problem reports, configuration status reports, as-coded design document, test results for archive, performance data, product measures, and process measures.

6. End Integration and Acceptance Test Phase, Start Operations and Maintenance Phase

   This endpoint is reached by completion of the Acceptance Test activity. The product is delivered to the user and installed. The product may include: documentation, reports, development standards, support tools used to develop the software, development data, test results and procedures as well as the code. Training may be provided as well as any warranty service.

   - Activities - Installation, support, training.

   - Outputs - All deliverables.

<u>2.1.3 Using Metrics During Software Development</u>

The Software Quality Framework has provided a language and structure for identifying software quality goals as well as providing some metrics of software product and process criteria. The previous section established start and endpoints for phases within the software process life-cycle and listed various products, development data, and process attributes which can potentially be evaluated by some measurement tool based on the ideas stated in the Software Quality Framework. A subset of the phases, products, and applicable metrics is summarized in Table 1.

Table 1. Summary of Phases, Products, and Metrics

Life Cycle Activity	requirements	architectural design	detailed design	code and unit test
Products	requirements document	architectural design document	detailed design document	code
Metrics	Halstead # pages	McCabe Halstead # lines McCall	McCabe Halstead # lines McCall	McCabe Halstead # lines McCall

2.2 SELECTING METRICS FOR AUTOMATIC MEASUREMENT DURING DESIGN

As part of Task 4.1.1, Develop Design Phase Software Quality Metrics, a survey of the current literature on software metrics was conducted. The goal of the search was to identify metrics which are valuable predictors of software quality and other software development parameters, and can be applied during the design phases. The results of the survey are presented in Section 2.2.1, and the definitions of the identified metrics are presented in Sections 2.2.2 and 2.2.3.

Of the metrics identified during the survey, those which were compatible with the Software Quality Framework were evaluated along with the Framework metrics for application during the design phases and automated data collection. Section 2.2.4 discusses the criteria that make a metric suitable for use during the design phases, and the conditions under which it may be measured by a design tool. The detailed assessment of the McCall, McCabe, and Halstead metrics follows in Section 2.2.5.

2.2.1 A Survey of Recent Metrics Literature

The sources for the metrics literature survey are included in the Bibliography. They show that much of the work going on with software metrics today is in measuring existing code with the Halstead or McCabe metrics and using the values to predict quality parameters such as the number of errors. The predictions are then tested against actual error data, and the measurement techniques are refined. The results indicate that the various Halstead metrics and the McCabe cyclomatic complexity metric are useful over different types of applications and implementation languages. These are, then, prime candidates for use during the design phases.

The McCabe and Halstead metrics basically measure different aspects of the complexity of software: its structure and language use. In the Software Quality Framework developed by McCall et al., the quality criterion simplicity (the inverse of complexity) contributes to many of the quality factors, including testability, reliability and maintainability. As such, it is a valuable indicator of quality during the design phases. The Software Quality Framework, though, covers other quality factors and provides metrics to measure the criteria that affect them. To make sure that as many quality factors as possible were covered during the design phases, each McCall metric was considered in detail along with the McCabe and Halstead metrics to see which components might be measured during the design phases, and which might be automated.

2.2.2 McCabe's Cyclomatic Complexity Metric

McCabe's cyclomatic complexity metric is based on the desire to limit the size of modules in a software system so that they are easy to test and maintain. He proposed that the number of paths through a module is a better measure of testability and maintainability than just the number of instructions/statements in a module: A strictly sequential module, with just one path, may be easier to test and maintain than one with a more complicated control structure, even though the sequential module is longer. More paths through a module require more tests, and McCabe inferred that more paths will make it harder to locate and fix errors in the module or modify it.

The metric is based on the complexity of the control structure of a module and of the system of modules, so it is applicable during the design phase as soon as that structure begins to evolve.

2.2.2.1 Cyclomatic Complexity Metric Definition

The definition of the metric is based on the following graph theory.

A directed graph can be drawn to represent the control flow of any module. It resembles a flowchart, but is constructed with more formal rules. Each group of statements which is executed without a control transfer becomes a node, and the control transfers become arcs linking the appropriate nodes. Ideally, all possible paths through a module would be tested. In practice, this is often prohibitively expensive or impossible. (If there is a backward branch in a module, the number of potential paths is infinite). For a module with a single entry point and a single exit point, a graph theory result applies: a basis for the graph can be found, in terms of basic paths, which, in linear combinations, generate all possible paths through the graph. McCabe proposes that the number of paths in the basis is a reasonable measure of the testability and maintainability of a module. It can be used to decide when to break up a module, or to identify modules which will be more difficult to test and maintain.

The number of basic paths in a graph, the cyclomatic complexity, is defined in [McC 76] as

$$v = e - n + 2p$$

where

n is the number of nodes in a directed graph of the program,
e is the number of edges (arcs) in the graph,
p is the number of connected subcomponents (modules) of the program,
and
v is the number of basic paths through the module.

If v is calculated for all the modules in a system, the value of the metric for a system will be the sum of the values for the modules in the system.

The cyclomatic complexity can also be obtained from a visual inspection of the program graph, when the graph is planar and connected. In this case, the cyclomatic complexity is equal to the number of regions into which the graph divides the plane.

Finally, for structured programs, the cyclomatic complexity is equal to the number of predicates (binary decisions) in the graph plus one. (Compound conditions and multi-way branches are treated as if they were the equivalent set of nested single-condition binary branches). A graph with no branch points leaves the plane in one region. Each binary decision adds another region to the graph.

McCabe recognized that more than one simple predicate could be included in a single programming statement, such as an IF. Predicates beyond the first clearly add complexity, but the graph would only show a single binary branch. To account for the added complexity, he proposed counting the number of simple predicates, rather than the number of decision statements. This amounts to modelling any compound condition as a set of nested IF statements.

Myers [Mye 77] proposed that the metric value should be an interval, rather than a single number. The lower bound would be the number arrived at by using the number of decision statements, and the upper bound would be the number arrived at by using the number of simple predicates. This method has the satisfying effect of giving the metric values for simple examples the same ordering as their subjective evaluations of complexity.

Neither McCabe's method nor Myers' method gives different weight to conditions at different nesting levels.

2.2.3 Halstead's Software Science Metrics

Halstead's Software Science [Hal 77] is a theory which has been applied to the complex problem of software project management. Halstead claimed that algorithms expressed as computer programs or other written media could be analyzed in a consistent and simple manner to yield indicators or measures of quality and other software development parameters. Researchers over the years have used this theory in a variety of experiments which, in general, has shown it to have some degree of utility. Most experiments to date have, however, used this technique on software at the code stage of its development. This task extends some previous work [Szu 80, Szu 81] which claimed that useful software science measures could be derived from software design media by embedding this metric technique in an automated design-aid tool.

2.2.3.1 Software Science Definitions

The Halstead technique is based on the identification and enumeration of four basic parameters that are directly available from the language used to express the algorithm. Other parameters, which are derived from these, form the set of software science metrics. All of the parameters are shown in Table 2.

The four basic parameters are:

η_1	number of distinct operators
η_2	number of distinct operands
N_1	total number of operators
N_2	total number of operands.

According to Halstead, algorithms consist of operators and operands, and nothing else. The validity of this claim is apparent when programming languages are considered, but when other forms of algorithm representation are used, ambiguities often arise. Languages which have structured or abstract data types generally cause the most difficulty. It is up to the user of the

Halstead technique to decide which elements of the vocabulary are operators and operands according to the vague definitions provided. Operators are symbols or combination of symbols which affect the value or ordering of operands. Operands are variables or constants that the implementation employs.

2.2.3.2 A Generalized Halstead Technique

When considering adapting the Halstead technique to a particular design medium, the following procedure is offered.

1. Identify operators and operands from the vocabulary of the language used in expressing the design. This step is often the most crucial and controversial part in the technique. Much prior work (e.g., [Els 78], [Fit 78], [Ham 82], [She 83]) has established evidence that it is difficult (and sometimes impossible) to determine whether a vocabulary element is to be classified an operator or operand. There is no consistent and non-ambiguous definition to use as a foundation. This has caused substantially different results by different researchers. It is therefore important to be aware of these findings and be consistent within an experimental domain.

2. Count the number of occurrences of each operator and operand. Again, this step needs to be consistent. Halstead, for example, proposed that each "GO TO label" be counted as a unique operator for each unique label, yet he considered "IF statements" as n occurrences of one unique IF operator.

3. Calculate the metrics based on the formulas listed in Table 2.

In this report, consistent counting techniques are defined for the DARTS design medium (Section 2.3.2), and for Ada as a design medium (Section 2.5.4). Other examples can be found in the literature. Most, however, consider only programming languages like Fortran, PL/I, and IBM assembly language.

Table 2. Halstead's Software Science Metrics.

HALSTEAD METRIC	SYMBOL	FORMULA
DISTINCT OPERATORS	η_1	
DISTINCT OPERANDS	η_2	
TOTAL OPERATORS	N_1	
TOTAL OPERANDS	N_2	
VOCABULARY	$\eta = \eta_1 + \eta_2$	
DESIGN LENGTH	$N = N_1 + N_2$	
ESTIMATED LENGTH	$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$	
PERCENT OFF		
DESIGN VOLUME	$V = N \log_2 \eta$	
POTENTIAL VOLUME	$V^* = \eta^* \log_2 \eta^*$	
DESIGN LEVEL	$L = V^*/V$	
ESTIMATED DESIGN LEVEL	$\hat{L} = 2\eta_2/\eta_1 N_2$	
INTELLIGENCE CONTENT	$I = \hat{L}V \approx V^*$	
LANGUAGE LEVEL	$\lambda = LV^*$	
ESTIMATED LANGUAGE LEVEL	$\hat{\lambda} = \hat{L}^2 V$	
EFFORT	$E = V/L$	
ESTIMATED EFFORT	$\hat{E} = V/\hat{L}$	

2.2.3.3 Interpreting Software Science Metrics

Software science metrics measure various complexity aspects of software components. The following discussion provides an interpretation of some of the more usable metrics defined in Table 2.

The Length (N) is roughly equivalent to the conventional count of executable statements in a program. This is based on observed occurrences of operators and operands.

The Estimated Length ($\hat{N}$) is a metric used to predict the ideal length of an implementation based only on the number of unique operators and operands available in the language used. This metric is often compared with the observed length to determine whether impurities are present in the implementation. Impurities generally reflect poor programming practice and make the implementation less concise than ideal.

The Volume (V) represents the number of bits necessary to encode the program using one character per operator or operand.

The Potential Volume (V*) is the most compact form the program could take, considering it as a built-in function with a list of input and output arguments. This term is often based on opinion, rather than calculation.

The Level (L) is a ratio between the Potential Volume and the actual Volume. It is often regarded as an indicator of the level of abstraction of implementation. Since the Potential Volume is an ambiguous quantity, the Estimated Level ($\hat{L}$) is often used, for it is dependent on readily observable quantities.

The Intelligence Content (I) is an estimate of the Potential Volume. It is independent of the language used and is expected to be invariant over different implementations of the program.

The Language Level (λ) measures the expressive power of a particular language. High level languages, that allow alternatives for expression, have language levels greater than those of assembly language. Halstead measured a variety of languages and computed the following: English prose = 2.16; PL/I = 1.53; FORTRAN = 1.14; and Assembly = .88. The Language Level is often estimated ($\hat{\lambda}$) to avoid using the questionable Potential Volume term.

The Effort measure (E) is used to predict total programming time when divided by the Stroud Number (number of elementary mental discriminations per second by a human). Halstead also provided a formula for estimating Effort ($\hat{E}$) which provides an alternative form. This alternative is discussed in more detail in Section 3.1.1.

2.2.4 Distinguishing Metrics by Phase and Automation Potential

This section establishes the potential for automatically measuring the McCall [McC 79], McCabe, and Halstead metrics in the early life-cycle phases, using a design tool such as DARTS.

McCall broke down the software development process into requirements, design, and implementation phases to indicate when the metrics could be applied. Later versions of the Framework [Boe 83a] [Boe 83b] divided the design phase further, into the Top Level and Detailed design subphases, but did not reclassify the metrics. This life-cycle phasing is more appropriate for this discussion because the subphases mirror the breakdown into preliminary and critical design reviews, and architectural and detailed design documents, as discussed in Section 2.1, providing measurable products and processes.

The criterion which determines whether a metric is applicable to a phase or subphase is that the information required for its evaluation be present in the software product for that phase or subphase.

For the requirements phase, the product is the Software Requirements Specification; for the design subphases, the Software Architectural Design Document and the Software Detailed Design Document. For the implementation phase, the product is the source code.

For the design phase, similar information is contained in the documents for both subphases, but the level of detail differs. The Proposed Military Standard on Defense System Software Development, (MIL-STD-SDS) [DoD 82], outlines the required documents. The information called for in the design document descriptions includes requirements allocations, interrupts, timing and sizing estimates and budgets, limitations and constraints, special requirements, interface definitions, database definitions, and control structure and flow. The architectural (Top Level) design document defines these items for the Computer Software Configuration Item (CSCI), shows the structure to the Computer Software Component (CSC) level, and defines the required items for each CSC.

The detailed design document evolves from the architectural design document. When it is completed, it includes a full breakdown of each CSC into units and modules, to the level required for coding. All the control logic is shown, and the data structures are broken down to individual variables. The items listed above are included for each module. (The detailed design subphase could be divided in two to yield three subphases for the design phase. The first part of detailed design would have a product in which the breakdown into units and modules is shown, and the second would have a product in which the logic and data internal to each module is shown. The former would show the control structure for the whole CSCI, while the latter would provide the code-to specification in a form easy to parcel out.)

Whether or not a metric can be measured by a design representation tool depends not only on the information needed to evaluate it being present, but also on its being present in a form which can be recognized and manipulated by a program. Generally, this involves providing a syntactically recognizable form for the information. For example, in DARTS, the input and output data items for a node are specified separately from the processing description, the inputs are distinguished from the outputs, and each item is distinct. This structure makes its possible to answer questions like:

1. Are any data items specified for a particular node?

2. How many data items does the node have?

3. How many input items does the node have?

4. How many of the input items are also output items?

This does not, however, allow us to answer the question,

5. Are all input items used in the node?

since the answer depends on other information about data use being present in the database and in a form that can be processed. As an alternative, it would have been possible to have provided a field for data items where input and output were not differentiated. The user might have included comments distinguishing input and output. The information content of the database would be the same, but the capability for automatically answering the questions would not: Questions 1 and 2 could be answered, but not 3-5. Finally, it would have been possible to have provided no separate data item field. The user might then have included the data item information in a free form text field associated with the node. Again, the information content would be the same, but in this case none of the questions could be answered automatically.

Even when the necessary information is present, some metrics are inherently not measurable by a design *representation* tool, because they involve a knowledge of the problem, and a human judgment about the sufficiency or completeness of the design. Most of the McCall metrics are really checklist items, such as making sure that all device errors are handled ([McC 79] ET.5(2)), or that the numerical methods used are sufficient (AC.1(4)). A tool which has a design representation as its database cannot tell whether something has been left out of that representation, unless it can do so by checking consistency with another source of information in the database. It could contain a list of such items and let the user check them off. The metrics could be derived from such a list.

In this project, DARTS is used as the tool context in which the metrics are evaluated for potential of automatic measurement. The context for information content is based on the design documents previously mentioned. Although most of the information called for in the design documents can be included in a DARTS database, much of it can be expressed only as comments in the free-form text associated with a node. Shown in Table 3 and Table 4 is how the information in each paragraph of the MIL-STD-SDS design documents can be represented in DARTS. The paragraph numbers are taken from the associated Data Item Descriptions (DIDs) ([DoD 82] R-DID-110 and R-DID-111).

In addition to the presence or lack of information, the feasibility of measuring a metric automatically depends on the practicality of performing the necessary processing. This may range from trivial to beyond the scope of the existing technology.

Table 3. MIL-STD-SDS Top Level Design Document Information in DARTS

Paragraph number	Means of representation in DARTS
1.1	Text.
1.2	Text.
2.1	Text.
2.2	Text.
3	Text, requirements and architectural design trees.
3.1	Architectural design tree, data set/use table, module table.
3.1.1	Text or by modelling the data source as a data producing process.
3.1.2	Text or by modelling the data source as a data producing process.
3.2	Text.
3.3.1	Text or could be modelled.
3.3.2	Diagram, ECSL simulation output.
3.3.3	Diagram.
3.3.4	DUR statement in tab.
3.4	Same diagram as 3.3.
3.4.X	Text.
3.4.X.1	Text.
3.4.X.2	Indata, Outdata; data set/use table.
3.4.X.3	Text.
3.4.X.4	Text or by modelling a handler.
3.5.all	There are no DARTS facilities naturally suited for representating data structures.
3.6.all	Text.
4, 5, 7-9	Intentionally left blank in DIDs.
6	Text.
10	Appendices - not applicable.

Table 4. MIL-STD-SDS Detailed Design Document Information in DARTS

Paragraph number	Means of representation in DARTS
1.1	Text.
1.2	Text.
2.all	Text.
3	Text.
3.1	Database printout.
3.1.1	Diagram, #variables.
3.1.1.1	Text; requirements, architectural design, and detailed design trees.
3.1.1.2	Text. Timing through DUR statements in tabs, output of ECSL simulation.
3.1.1.3	Diagram except data structures. Data descriptions in text.
3.1.1.3.Y a)	Text.
3.1.1.3.Y b)	Data variables - partial.
3.1.1.3.Y c)	Diagram, indata, outdata.
3.1.1.3.Y d)	#variables, data set/use table.
3.1.1.3.Y e)	INV
3.1.1.3.Y f)	Diagram.
3.1.1.3.Y g)	Diagram.
3.1.1.3.Y h)	Data set/use table.
3.1.1.3.Y i)	Text.
3.2.1.X.6	May be modelled.
3.2.1.X.7	Text.
3.2.1.X.8	DUR statements in tabs.
3.2.1.X.9	Text.
3.2.1.X.10	Text.
3.2.1.X.11	Diagram.
3.2.1.X.12	May be modelled, DUR statements in tabs.
3.3	Text.
4, 5, 7-9	Intentionally left blank in DIDs.
6	Text.
10	Appendices - not applicable.

2.2.5 Evaluation of Candidate Metrics

Table 5 summarizes the results of evaluating each of the metrics in [McC 79] Appendix B, the McCabe cyclomatic complexity metric [McC 76], and the various Halstead metrics [Hal 77]. The metrics were assessed against the criteria discussed above in deciding during which software development phase they are applicable, and whether or not they may be measured by using the information in a DARTS database.

The table shows [McC 79]'s designation of applicability for the design phase in the center column under design (labeled McC), between the designations for the architectural (labeled AD), and detailed (labeled DD) design subphases. When the designation for any phase disagrees with [McC 79], it is appended with an "*".

The criteria used to determine each metric's potential for automation, noted in the AUTO column of the table, are relative to the current implementation of DARTS. The following symbols are used to indicate the distinctions discussed in the section "Distinguishing Metrics by Phase and Automation Potential."

Y0 - Use of DARTS guarantees that this metric takes a particular value. For example, the metric MO.2(1) asks how much of the design is represented in a hierarchical structure. DARTS representations are inherently hierarchical, so the metric always takes the value 1.

Y1 - The information needed is present in a form recognizable by a program, so the metric is automatable.

N1a - The information needed is not present, but might be added.

N1b - The information needed is not present, and cannot be added.

N1 - The information needed is not present, and is not really appropriate in a design representation tool.

Y2 - The information needed is present, but not in a recognizable form. DARTS could be changed to provide a syntactic form for the information.

N2 - The information needed is present, but not in a recognizable form. DARTS cannot be changed to provide a syntactic form for the information.

N3 - The information needed is present, but the metric is not amenable to automatic calculation. It asks a question which requires understanding of the problem being expressed, for example, whether some of the processing expressed in the design is complete, sufficient, or correct; or whether a particular function is present in the design.

The practicality of providing the necessary processing is indicated in the table in a gross fashion. An "*" is appended to the automatability symbol if the processing is likely to involve a substantial effort. "Note" follows the table entry if a note is provided in the following section to comment on or explain the entry.

2.2.5.1 Notes on Table 5

CP.1(1) This metric asks whether or not all references are unique, e.g., a function is not called by one name in one place and another somewhere else. This is really a completeness question. It is not determinable automatically, since the processing will assume that different names refer to different entities. However, the data set/use table and the module table present the information needed in a tabular form so that the judgment may be more easily made. For example, a spelling error should be obvious.

CP.1(5) This metric asks whether or not all conditions and processing are specified for each decision point. This could be enforced if DARTS were modified to require that selector nodes have n+1 offspring for n predicates, instead of assuming a null n+1st offspring.

CP.1(8,9) These metrics check consistency between requirements and design, and design and code. [McC 79] gave this up as too difficult to measure.

CS.2(2) This metric is the proportion of modules which do not conform to naming conventions. Naming conventions should be used from the beginning, in requirements definition, to help traceability and prevent confusion. The "N1a" index here is used to indicate that a particular convention could be added; the "N3", that the metric relies on a human judgment.

CS.2(3) This metric calls for global data items to be defined in the same manner in all modules, so it is another example of measuring conformity to a convention. Insofar as global items may be defined in DARTS, the checknodes function detects anomalies.

CS.2(5) This metric measures data type consistency. A strongly typed implementation language would enforce this.

ET.1(1) This metric asks whether or not any concurrent processing is centrally controlled. Since DARTS provides a means for expressing processing concurrency and data exchange, any concurrent design expressed in it will be consistent. The metric definition does not contain enough detail to know if this is what is meant, or if it is a checklist item.

Table 5. Metric Applicability and Automatability (Part 1 of 6)

METRIC	REQTS	DESIGN AD	DESIGN McC	DESIGN DD	IMPL	AUTO	
TR.1 Cross reference relating modules to requirements	N	Y	Y	Y	Y	N1a	
CP.1 COMPLETENESS CHECKLIST							
(1)	Y	Y	Y	Y	Y	YO/N3	Note
(2)	Y	Y	Y	Y	Y	N1a	
(3)	Y	Y	Y	Y	Y	Y1	
(4)	Y	Y	Y	Y	Y	Y1	
(5)	Y	Y	Y	Y	Y	Y1	Note
(6)	N	N	Y	Y	Y	N1a	
(7)	Y	Y	Y	Y	Y	N1a	
(8)	N	Y	D1	Y	N	-	Note
(9)	N	N	D1	N	Y	-	Note
CS.1 PROCEDURE CONSISTENCY MEASURE							
(1)	N	Y	Y	Y	Y*	YO	
(2)	N	N	Y	Y	Y	YO	
(3)	N	N	Y	Y	Y	N3	
(4)	N	N	Y	Y	Y	N3	
CS.2 DATA CONSISTENCY MEASURE							
(1)	N	N	Y	Y	N*	N1a	
(2)	Y*	Y	Y	Y	Y	N1a/3	Note
(3)	N	Y	Y	Y	Y	N1a/3	Note
(4)	N	N	D1	Y	Y	-	
(5)	N	N	D1	Y	Y	-	Note
AC.1 ACCURACY CHECKLIST							
(1)	Y	N	N	N	N	-	
(2)	Y	N	N	N	N	-	
(3)	N	N	Y	Y	N	N3	
(4)	N	N	Y	Y	Y	N3	
(5)	N	N	N	N	Y	-	
ET.1 ERROR TOLERANCE CTL CHECKLIST							
(1)	N	Y	Y	Y	Y	YO/N3	Note
(2)	N	N	Y	Y	Y	N3	
(3)	N	N	Y	Y	Y	N1a	

Table 5. Metric Applicability and Automatability (Part 2 of 6)

METRIC	REQTS	---- DESIGN ----			IMPL	AUTO	
		AD	McC	DD			
ET.2 RECOVERY FROM IMPROPER INPUT							
(1)	Y	N	N	N	N	-	
(2)	N	Y	Y	Y	Y	N3	
(3)	N	Y	Y	Y	Y	N3	
(4)	N	Y	Y	Y	Y	N3	
(5)	N	Y	Y	Y	Y	N3	
ET.3 RECOVERY FROM COMP. FAILURE							
(1)	Y	N	N	N	N	-	
(2)	N	N	Y	Y	Y	N1a*	Note
(3)	N	N	Y	Y	Y	N1a*	Note
(4)	N	N	Y	Y	Y	N3	
ET.4 RECOVERY FROM HARDWARE FAULTS							
(1)	Y	N	N	N	N	-	
(2)	N	Y	Y	Y	Y	N3	
ET.5 RECOVERY FROM DEVICE ERRORS							
(1)	Y	N	N	N	N	-	
(2)	N	Y	Y	Y	Y	N3	
SI.1 DESIGN STRUCTURE MEASURE							
(1)	Y*	Y	Y	Y	Y	Y0	
(2)	N*	N	N	Y	Y*	N3	
(3)	N*	N	N	Y	Y	N3/Y2*	Note
(4)	N*	Y*	N	Y*	Y	N3	
(5)	N	N	N	Y*	Y	Y0	
(6)	N	N	Y	Y	Y	Y0/N1a	Note
(7)	N	N	Y	Y	Y	N1a	Note
(8)	N	Y	D1	Y	D1	-	
(9)	N	Y*	N	Y*	D2	-	
SI.2 USE OF STRUCTURED LANGUAGE OR PREPROCESSOR	N	N	N	N	D2	-	Note
SI.3 COMPLEXITY MEASURE	N	Y	Y	Y	Y	Y2	Note
SI.4 MEASURE OF CODING SIMPLICITY							
(1)	N	N	N	N	Y	_	
(2)	N	N	N	N	Y	_	
(3)	N	N	N	N	Y	_	
(4)	N	N	N	N	Y	-	
(5)	N	N	N	N	Y	-	
(6)	N	N	N	N	Y	-	
(7)	N	N	N	N	Y	-	
(8)	N	N	N	N	Y	-	
(9)	N	N	N	N	Y	-	
(10)	N	N	N	Y*	Y	Y1	
(11)	N	N	N	Y*	Y	Y2	
(12)	N	N	N	N	D3	-	
(13)	N	N	N	N	D1	-	
(14)	N	N	N	N	D2	-	
(15)	N	N	N	N	D2	-	
(16)	N	N	N	N	D1	-	

Table 5. Metric Applicability and Automatability (Part 3 of 6)

METRIC	REQTS	---- DESIGN ----			IMPL	AUTO	
		AD	McC	DD			
MO.1 STABILITY MEASURE - applicable			D1				
MO.2 MODULAR IMPLEMENTATION MEASURE							
(1)	N	Y	Y	Y	Y	YO	
(2)	N	N	N	N	Y	-	
(3)	N	N	Y	Y	Y	N1a	
(4)	N	Y	Y	Y	Y	Y1	
(5)	N	Y	Y	Y	Y	Y1	
(6)	N	Y	Y	Y	Y	YO	
(7)	N	N	Y	Y	Y	N3	
(8)	N	Y	D1	Y	D1	N3	
GE.1 EXTENT TO WHICH MODULE IS REFERENCED BY OTHER MODULES	N	Y	Y	Y	Y	Y1	
GE.2 IMPLEMENTATION FOR GENERALITY							
(1)	N	N	Y	Y	Y	N3	
(2)	N	N	Y	Y	Y	N3	
(3)	N	Y	Y	Y	Y	N3	
(4)	N	Y	Y	Y	Y	N3	
(5)	N	Y*	N	Y*	D2	N1a	
EX.1 DATA STORAGE EXPANSION MEASURE							
(1)	N	Y	Y	Y	Y	N1	
(2)	N	N	N	N	Y	-	
EX.2 EXTENSIBILITY MEASURE							
(1)	N	Y	Y	Y	Y	N3	
(2)	N	Y	Y	Y	Y	N3	
(3)	N	N?	N	N?	Y	-?	Note
IN.1 MODULE TESTING MEASURE							
(1)	N	Y	Y	Y	Y	N1a	
(2)	N	Y	Y	Y	Y	N1a	
IN.2 INTEGRATION TESTING MEASURE							
(1)	N	Y	Y	Y	Y	N1	
(2)	N	Y	Y	Y	Y	N1	
IN.3 SYSTEM TESTING MEASURE							
(1)	N	Y	Y	Y	Y	N1	Note
(2)	N	Y	Y	Y	Y	N1	Note

Table 5. Metric Applicability and Automatability (Part 4 of 6)

METRIC	REQTS	---- DESIGN ----			IMPL	AUTO	
		AD	McC	DD			
SD.1 QUANTITY OF COMMENTS	N	N	N	N	Y	-	
SD.2 EFFECTIVENESS OF COMMENTS							
(1)	N	N	N	N	Y	-	
(2)	N	N	N	N	Y	-	
(3)	N	N	N	N	Y	-	
(4)	N	N	N	N	Y	-	
(5)	N	N	N	N	Y	-	
(6)	N	N	N	N	Y	-	
(7)	N	N	N	N	Y	-	
SD.3 DESCRIPTIVENESS OF IMPLEMENTATION LANGUAGE MEASURE							Note
(1)	N	N	N	N	Y	-	
(2)	N	N	N	N	Y	-	
(3)	N	N	N	N	Y	-	
(4)	N	N	N	N	Y	-	
(5)	N	N	N	N	D3	-	
(6)	N	N	N	N	D1	-	
EE.1 PERFORMANCE REQUIREMENTS IDENT. AND ALLOCATED TO DESIGN	Y	Y	Y	Y	N	N1	Note
EE.2 ITERATIVE PROCESSING EFFICIENCY							
(1)	N	N?	Y	Y	Y	N1/2	Note
(2)	N	N	N	N	Y	-	
(3)	N	N	N	Y*	Y	Y2*	
(4)	N	N	Y	Y	Y	N1	
(5)	N	N	Y	Y	Y	N2	
(6)	N	N	N	N	Y	-	
(7)	N	N	N	N	Y	-	
(8)	N	N*	Y	N*	Y*	-	Note
(9)	N	-	D1	-	Y	-	
(10)	N	-	D1	-	Y	-	
EE.3 DATA USAGE EFFICIENCY MEASURE							
(1)	N	N	Y	Y	Y	N1b	
(2)	N	N	N	N	Y	-	Note
(3)	N	N	N	N	Y	-	Note
(4)	N	N	N	N	Y	-	Note
(5)	N	N	Y	Y	Y	N1b	Note
(6)	N	N	N	Y*	Y	Y2*	Note
(7)	N	N	N	Y*	Y	Y2*	Note

Table 5. Metric Applicability and Automatability (Part 5 of 6)

METRIC	REQTS	---- DESIGN ----			IMPL	AUTO	
		AD	McC	DD			
SE.1 STORAGE EFFICIENCY MEASURE							Note
(1)	N	Y	Y	Y	N	N1a	
(2)	N	Y	Y	Y	Y	N1b	
(3)	N	Y*	N	Y*	Y	N3	
(4)	N	N	Y	Y	Y	N1	
(5)	N	Y	Y	Y	Y	N3	
(6)	N	N	N	Y*	Y	N3	
(7)	N	N	N	N	Y	-	
(8)	N	N	N	N	D1	-	
(9)	N	-	D1	-	D1	-	
(10)	N	-	D1	-	D1	-	
(11)	N	N	N	N	D1	-	
AC.1 ACCESS CONTROL CHECKLIST							Note
(1)	Y	Y	Y	Y	Y	N3	
(2)	Y	Y	Y	Y	Y	N3	
(3)	Y	Y	Y	Y	Y	N3	
AA.1 ACCESS AUDIT CHECKLIST							Note
(1)	Y	Y	Y	Y	Y	N3	
(2)	Y	Y	Y	Y	Y	N3	
OP.1 OPERABILITY CHECKLIST							
(1)	Y	Y	Y	Y	Y	N3	
(2)	Y	Y	Y	Y	Y	N3	
(3)	Y	Y	Y	Y	Y	N3	
(4)	N	N	N	N	Y	-	
(5)	N	Y	N	Y	Y	N3	Note
(6)	N	Y	Y	Y	Y	N3	
(7)	N	Y	Y	Y	Y	N3	
TN.1 TRAINING CHECKLIST							
(1)	N	Y*	N	Y*	Y	N3	Note
(2)	N	Y*	N	Y*	Y	N3	Note
(3)	N	Y	Y	Y	Y	N3	
CM.1 USER INPUT INTERFACE MEASURE							
(1)	N	N	Y	Y	Y	N1a	
(2)	N	N	Y	Y	Y	N1a*	
(3)	N	N	Y	Y	Y	N1b	
(4)	N	Y	Y	Y	Y	N3	
(5)	N	N	Y	Y	Y	N3	
(6)	Y	Y	Y	Y	Y	N3	

Table 5. Metric Applicability and Automatability. (Part 6 of 6)

METRIC	REQTS	---- DESIGN ----			IMPL	AUTO	
		AD	McC	DD			
CM.2 USER OUTPUT INTERFACE MEASURE							
(1)	Y	Y	Y	Y	Y	N3	
(2)	N	Y	Y	Y	Y	N3	
(3)	N	N	Y	Y	Y	N3	
(4)	N	N	Y	Y	Y	N1a*	
(5)	N	N	Y	Y	Y	N3	
(6)	N	N	Y	Y	Y	N3	
(7)	Y	Y	Y	Y	Y	N3	
SS.1 SOFTWARE SYSTEM INDEPENDENCE							
(1)	N	N	Y	Y	Y	N1a	
(2)	N	N	Y	Y	Y	N1	
(3)	N	-	D3	-	D3	-	
(4)	N	-	D3	-	D3	-	
MI.1 MACHINE INDEPENDENCE MEASURE							
(1)	N	N?	Y	Y?	Y	N3	Note
(2)	N	N?	Y	Y	Y	N1/3	Note
(3)	N	N	N	Y	Y	N3	
(4)	N	N	N	Y	Y	N3	
CC.1 COMMUNICATIONS COMMONALITY							
(1)	Y	N	N	N	N	-	
(2)	N	Y	Y	Y	Y	N3	
(3)	N	Y	Y	Y	Y	N3	
(4)	N	Y	Y	Y	Y	N3	
DC.1 DATA COMMONALITY CHECKLIST							
(1)	Y	N	N	N	N	-	
(2)	N	Y	Y	Y	Y	N3	
(3)	N	Y	Y	Y	Y	N3	
CO.1 HALSTEAD'S MEASURE (LENGTH)	N	Y	N	Y	Y	Y1	
MCCABE'S CYCLOMATIC COMPLEXITY	N	Y		Y	Y	?	
HALSTEAD'S METRICS							
Number of distinct operators	N	Y		Y	Y	Y1	
Number of distinct operands	N	Y		Y	Y	Y1	
Number of total operators	N	Y		Y	Y	Y1	
Number of total operands	N	Y		Y	Y	Y1	
Vocabulary	N	Y		Y	Y	Y1	
Length	N	Y		Y	Y	Y1	
Difficulty (1/Length)	N	Y		Y	Y	Y1	
Volume	N	Y		Y	Y	Y1	
Effort	N	Y		Y	Y	Y1	
Program level	N	Y		Y	Y	Y1	
Language level	N	Y		Y	Y	Y1	

ET.3(2,3) These metrics ask whether or not all loop and multiple transfer index parameters and subscripts are range checked before use. A higher level implementation language may have these capabilities.

SI.1(3) This metric asks whether or not module processing is dependent on prior processing. Evaluation of it depends on understanding the content of the module, hence the "N3". The "Y2*" is to indicate that reuse of local variables could be checked by interpreting tab statements, but is a difficult job. Though the no memory principle is usually a good one to follow, there are cases, like state machines and filters, where it is not the method of choice.

SI.1(6,7) These metrics have to do with database characteristics. DARTS could be modified to support them by addition of data structures. The number of variables could be used for (6), the size of the database.

SI.2 This metric asks whether or not a structured language or preprocessor is used. As described, it is for the implementation phase. An analog could be developed for the design subphases.

SI.3 This metric measures data and control flow complexity. It depends on the feasibility of deriving a graph representation from the DARTS database.

SI.4(1-4,6) These metrics measure coding simplicity using various methods. Analogs could be developed for the design subphases.

EX.2(3) This metric is the percent of speed capacity uncommitted. Though [McC 79] show its application during implementation, it should be included in the design phase, since tight machine resources have a dramatic effect on design.

SD.1 This metric measures the quantity of comments. An analog could be developed for the design subphases.

SD.2 This metric measures the effectiveness of comments. An analog could be developed for the design subphases.

SD.3 This metric measures the descriptiveness of the implementation language. An analog could be developed for the design subphases.

EE.1 This metric asks whether or not the performance requirements are allocated to the design. If a document existed which tagged each requirement (say by paragraph number), the tags could be included in the requirement and design diagrams. A metric could then be devised, but it would have the problem of quantifying completeness, since the requirements to design allocation may be many-to-one, one-to-one, or one-to-many.

EE.2(1) This metric asks what proportion of non-loop-dependent computations are in loops. Some higher level language optimizing compilers will minimize this. There may be practical exceptions to this principle.

EE.2(8) This metric asks whether or not the storage facility is used efficiently. The definition given is too vague for an evaluation to be made, depending on "evaluation of the utility of the storage facility". Also, the table shows this in the design phase, but not in implementation, which is probably a mistake.

EE.3(2,3,4) These are data usage efficiency measures for the implementation. Analogs could be developed for the detailed design subphase. A higher level language with strong typing might detect/prohibit mixed mode expressions.

EE.3(5) This definition is not detailed enough for an evaluation to be made.

EE.3(6,7) These metrics ask about the numbers of static and dynamic data items. They may be applicable to the detailed design subphase, as well as implementation. The variables could be partitioned into those which are changed and those which are not, by interpreting the tab statements. The metric might be more suitably measured in a tool based on a compiler, since a compiler typically processes this kind of semantic information. It might be suitable for a PDL-like design tool.

SE.1 This metric is concerned with storage efficiency. It depends on a variable having a locus of definition, which might or might not be true in a design representation. If it is, the information necessary is present, but the metric requires knowledge of the meaning of the variables, and so, is not automatable.

AC.1 and AA.1 are further examples where the metric registers whether or not a particular functional cabability is present in the system.

OP.1(5), TN.1(1,2) These metrics deal with job set up and tear down procedures, and training material. Some of the specification called for in these two categories should start during design, so that user feedback may be obtained to influence design.

MI.1(1) This metric measures machine independence in terms of whether the programming language used is available on other machines. If the implementation language is chosen early in the lifecycle, the metric could be measured during the design phase.

MI.1(2) This metric has to do with limiting the number of I/O references in a module. "I/O reference" is not defined specifically enough to enable an evaluation.

2.2.6 Metric Automation Potential Summary

The totals for each category used in assessing automation potential are shown in Table 6 for the McCall metrics. The McCabe and Halstead metrics are considered separately, below. Out of roughly 100 McCall metric components, about 25% (29) are good prospects for measuring in DARTS (the Y0, Y1, N1a and Y2 categories).

Table 6. Automation Summary for McCall Metrics

Category	Total
Y0	5
Y1	8
N1a	14
N1a*	4
N1b	4
N1	8
Y2	2
N2	1
N3	57

There are a number of metrics which have as their values the proportion of modules which conform to a set of conventions: naming, standard representation for procedures, data, etc. They are mostly in the N3 category. The conventions themselves are left unspecified, since different ones may be suitable for different projects. A design tool would clearly be able to check for adherence to a particular convention.

Another class of metrics is composed of what are really checklist items, like making sure that all device errors are handled. They make up the largest part of the N3 category. The metric depends on an assertion by a person that a module meets a certain criterion. This checklist capability is a function separable from design representation and metric measurement. We should consider whether a tool, or part of a unified tool package, would be of use in this area. It could act as a prompter for the system developers, and a quality assurance audit tool. The metrics in this class could be measured from it.

A few other metrics have the purpose of promoting conservative code practice assuming that the final code will be in assembler language, such as loop index range checking. As indicated in the notes for the table, they might be unnecessary if a high level language were used.

The McCabe cyclomatic complexity metric may certainly be automated, since the DARTS primitives represent the structure programming control structures.

The Halstead metrics may all be automated, since they all depend on the basic counts of operators and operands, easily captured from many forms of design media.

The McCabe and Halstead metrics were chosen for use in the other tasks of the project. Many of the McCall metrics have some components which are suitable for automation in DARTS, and some which are not; which makes them of dubious use as a demonstration of automatic measurement. The McCabe and Halstead metrics present a coherent picture of the complexity of the software, which is one of the McCall metrics. In addition, the Halstead metrics have been shown to be of some use in predicting planning parameters.

2.3 DESIGN METRICS AND DARTS

The next sections present the software requirements and design for the McCabe and Halstead metric implementation in the CSDL design-aid tool DARTS.

2.3.1 McCabe's Cyclomatic Complexity Metric

This section includes the requirements and detailed design specification for the implementation of McCabe's cyclomatic complexity metric [McC 76] in DARTS.

2.3.1.1 Requirements

The McCabe cyclomatic complexity metric shall be implemented in DARTS. The McCabe module shall be in PL/I, fit into the existing software structure, use the existing database access routines, and use other existing utility routines where feasible. The user shall specify the top node and depth of the subtree to be measured. The output shall show which subtree was analyzed, and include the metric values for each distinct module and the total for the specified subtree.

Special Processing: DARTS uses a hierarchical representation technique which allows a user to truncate trees at any level. This is useful when it is advantageous to hide some details of a design to show only the higher levels. Truncation often leaves ambiguous iterator and selector nodes (decision nodes with no offspring). In these cases, if the tab for the iterator or selector has predicates in it, processing shall be as normal. If the tab has no predicates, an iterator shall be assumed to have a single simple predicate to terminate the loop. A selector shall be assumed to have a single simple predicate used to distinguish between two offspring (an IF-THEN-ELSE).

2.3.1.2 The DARTS Implementation of McCabe's Metric

Cyclomatic complexity may be measured from a DARTS representation of a design, since the DARTS primitives represent control flow. The definition based on counting the binary predicates is the most natural one to implement, since the non-real-time control structures in DARTS are the same as those used for structured programming.

The following figures show the DARTS tree representations of the common control structure primitives and explain their graph equivalents. The trees for component and exchange nodes are not shown since they are just single nodes: they do not represent any control structure.

The coordinator (Figure 2) represents parallel execution of two or more processes, so the equivalent graph would include each process as a separate unconnected component. As discussed below, the graph medium does not allow representation of control interactions brought about by timing relations and data exchanges, so coordinators will be treated as sequencers for this measurement.

The DARTS iterator (Figure 3) can be used to represent any kind of loop, including FOR, WHILE, or UNTIL loops. It may have one of two graph equivalents depending on whether the loop termination test occurs at the beginning or end of the loop. Any number of steps may occur inside the loop.

The selector (Figure 4) is used to select one from a group of alternatives. It may represent an IF, IF-THEN-ELSE, or CASE construct. If n-1 predicates are specified, an nth selection may be chosen if they are all false; but there need not be an nth selection. Its graph would include a multi-way branch or a series of nested binary branches.

The sequencer (Figure 5) just ties together nodes which are executed one after the other. It graph equivalent is linear. Any number of steps may occur in sequence.

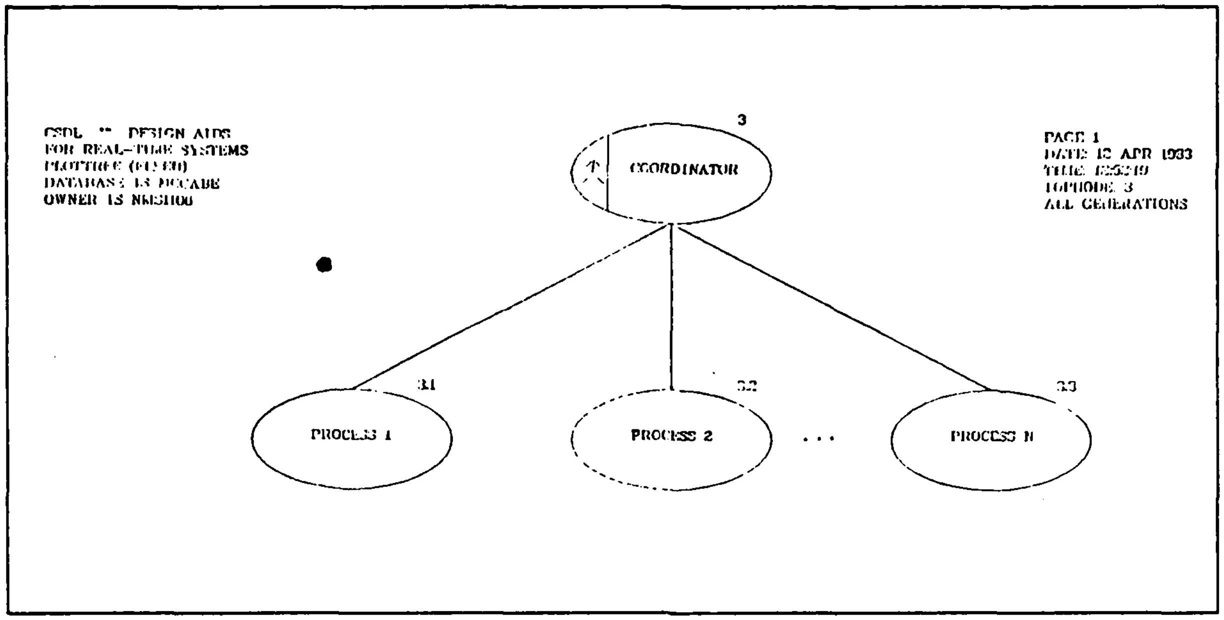


Figure 2. DARTS Tree for Coordinator

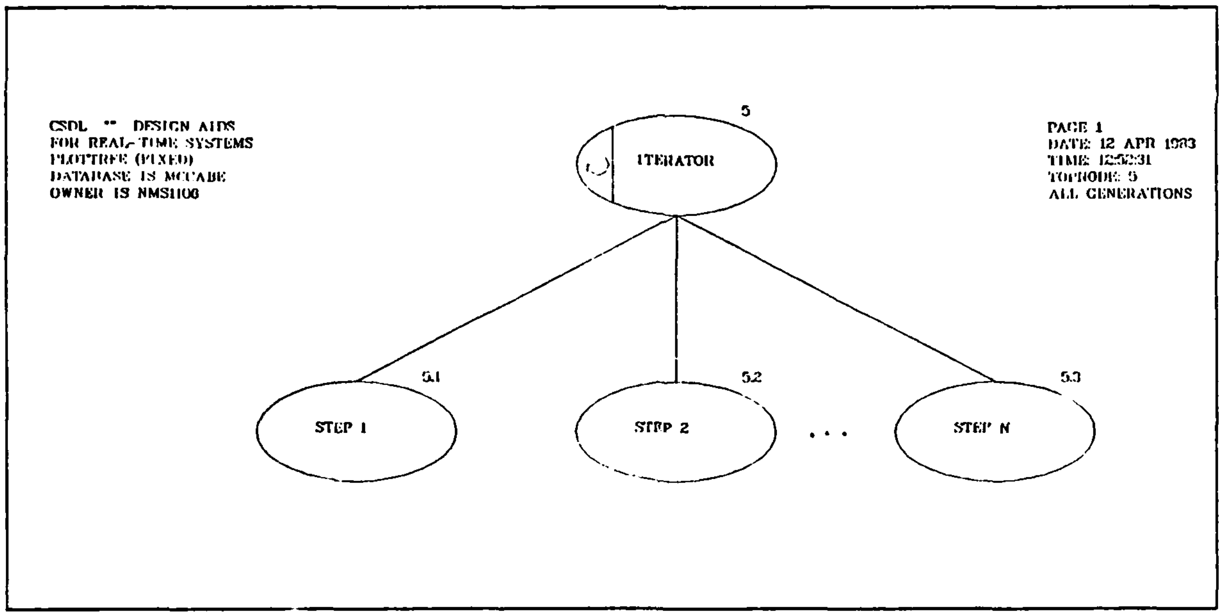


Figure 3. DARTS Tree for Iterator

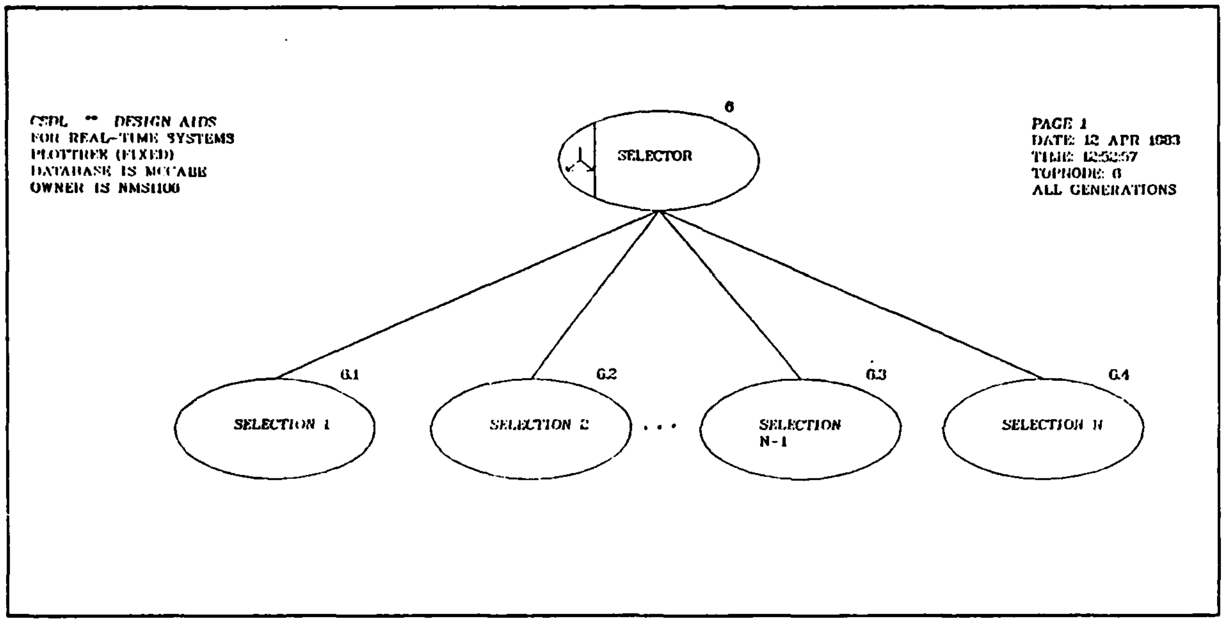


Figure 4. DARTS Tree for Selector

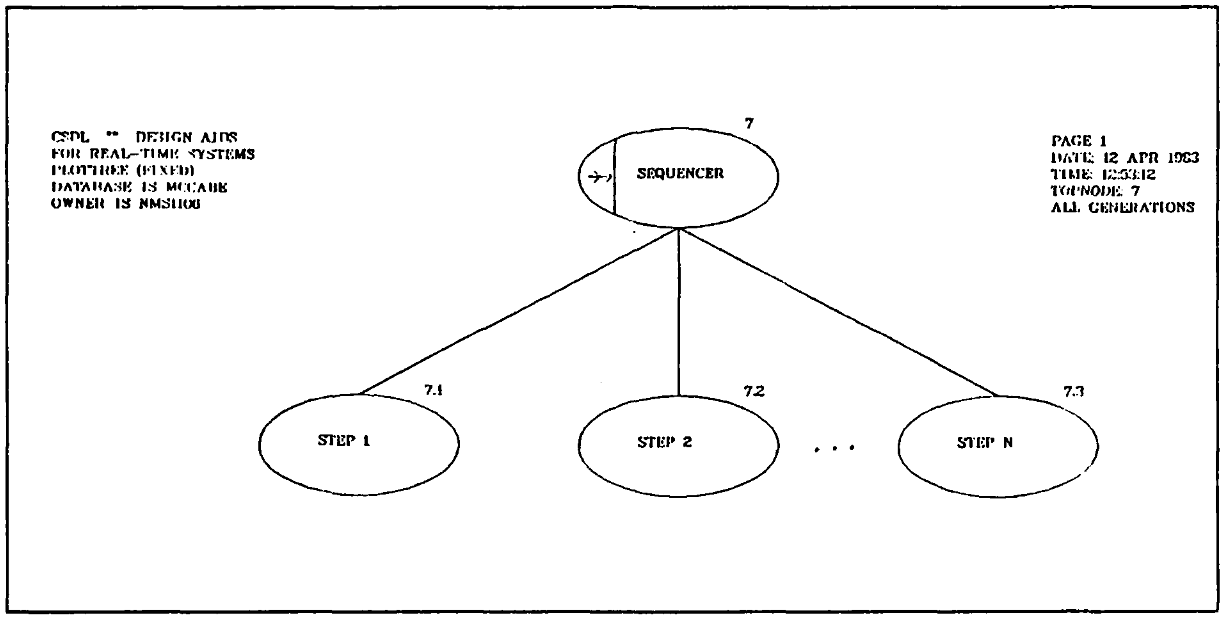


Figure 5. DARTS Tree for Sequencer

The rest of this section details how the metric is implemented in DARTS, ensuring that the criteria are met for application of the simplified definition based on counting decisions. That is, that the control graph for the DARTS tree is connected, planar, and has a single entry and exit. Points considered are:

1. the applicability of the metric to multi-process systems, and systems using the real-time data exchange construct,

2. the influence on the metric of the statements of the tab language,

3. how the number of modules in a system will be determined,

4. and how predicates will be counted for each type of DARTS node.

The cyclomatic complexity metric is not intended to measure complexity due to real-time aspects of a problem. Another metric must be used to cover this area. The real-time constructs, coordinator and exchange nodes, may have predicates implied in their implementation, just as the implementation of sequential control may; but to the designer, they are primitives. Exchange nodes are treated as component nodes and coordinator nodes are treated as sequencer nodes. Since the complexity due to timing requirements is not reflected in this metric, this is a reasonable approximation. See the discussion of BLK and SEG statements, below, for causing the processes under a coordinator to be handled as separate modules.

The general scheme for measuring cyclomatic complexity is to traverse a user-specified subtree, accumulating the number of decisions and simple predicates for each node. The number of decisions plus one yields the lower bound for the interval described by Myers, and the number of simple predicates plus one gives the upper bound. The value of the metric for each module encountered is printed, as well as the total for the subtree specified.

A software design in DARTS can be either abstract or detailed. The level of detail is represented by using both uninterpreted and interpreted node forms. Uninterpreted nodes are those which do not contain any primitive operators such as arithmetical or logical operators in a special tab area beneath the node (see example DARTS trees in the Appendices). Generally, nodes appearing near the root node of a process architecture tree are more likely to represent uninterpreted functions. Interpreted nodes are those which contain primitives such as arithmetic or logical operators in the tab area. These tab statements are used to indicate actions which occur at the node, conditions under which control transfers are made, or to further define the node.

The absence or presence of these tab statements makes a significant difference in calculating the McCabe metric values, so, two methods are used. For nodes without tab statements, the entire user-specified subtree is considered as one module, and the type of node and number of offspring determine the number of decisions and predicates. This method is useful for high-level designs, where the simple predicates in terms of actual variables and the software structure in terms of modules are often not known. For nodes with tab statements, the statements are inspected and used to determine the number of modules and the number of predicates. Details on the recognition of modules and the counting of predicates follow.

Modules in DARTS are indicated by two statements in tabs: BLK and SEG. Each statement assigns a name to a subtree. The SEG statement defines the subtree as a subroutine which may be invoked by an INV statement in a tab in another part of the system tree. (Without SEGs, commonly used subtrees must be repeated at each point of use).

When a BLK or SEG statement is present as the first statement in the tab, the subtree for that node is recognized as a module, and a separate count of decisions and predicates is made for it. Its metric value interval is shown separately in the output. Since a coordinator node is handled as if it were a sequencer, the processes under it must have BLK or SEG statements in their tabs if the processes are to be handled as separate modules.

A list of the modules invoked is compiled, and an output line referencing a footnote is output if the corresponding SEG is not present in the specified subtree. In this case, the totals shown for the user-specified subtree are incorrect, since the values for the missing SEGs are not included.

In the following specification for how the numbers of decisions and predicates are determined, "simple predicate" is used to indicate what McCabe calls a predicate, and "tab predicate" is used for a DARTS tab predicate, which may actually be a conjunction of several simple predicates.

For nodes without tab statements, each iterator node is assumed to have a single decision to terminate the loop. Each selector node is assumed to have a decision to reach each of its offspring except the last (the ELSE branch), so the number of decisions is the number of offspring minus one. If there are no offspring, the selector is assumed to have a single decision used to distinguish between two offspring (an IF-THEN-ELSE). All other node types have zero decisions. For all node types, the number of predicates is the same as the number of decisions.

For nodes with tab statements, all node types except iterators and selectors again have zero decisions and zero predicates.

Iterator nodes may or may not have a tab predicate specified in the tab text. Those with no tab predicate will be assumed to have a single unstated decision/predicate for terminating the loop, as for nodes without tab statements. For those with a tab predicate, the number of decisions is still one, but the number of simple predicates is the number of ANDs in the first tab predicate plus one. (The number of ANDs is one less than the number of simple predicates which they join; and ORs are not allowed). Since iterators are used to represent a number of different kinds of loops, including FORs, WHILEs and UNTILs, this method corresponds to mechanizing any kind of loop as a series of nested IF-THEN-ELSEs.

Selector nodes also may or may not have tab predicates specified in the tab statements. They are used to represent constructs such as IF-THEN-ELSE and CASE, so more than one tab predicate may be present. Those with no tab predicate will be treated as if there were no tab statements. For those with tab predicates, the number of decisions is the same as for selectors without tab statements. The number of simple predicates in each tab predicate is determined as it is for iterators, and the sum for all the tab predicates is the number of simple predicates for the node. Again, this reflects the mechanization of any selector as a series of nested IF-THEN-ELSEs.

A tab statement is recognized as a predicate according to the specifications in [CSDL82]. That is, if the statement does not begin with one of the reserved words (BLK, SEG, etc.), and it contains one of the relational operators EQ, NE, GT, GE, LT, or LE. The tab statement TRUE is also recognized as a predicate. Redundant or degenerate simple predicates, such as "X GT 3 AND X GT 3" or "TRUE AND X GT 3" are not detected. Handling of variable names which are reserved words is undefined.

2.3.2 Halstead's Software Science Metrics

This section specifies how the Halstead counting method, and the associated metric calculations are implemented in DARTS to assess the quality of software designs. It is expected that this metric analysis will provide designers and managers with useful feedback during software development.

2.3.2.1 Requirements

The Halstead parameters shown in Table 2 shall be measured in DARTS. Operators and operands shall be identified in the DARTS medium and a counting method shall be defined which is consistent with the definitions provided in by Halstead [Hal 77]. The Halstead module shall be written in PL/I, fit into the existing DARTS software structure, use the existing database access routines, and use other existing utility routines where feasible. The user shall specify the top node and depth of the subtree for which the parameters are to be measured, and the counting method to be used. The output shall show which subtree was measured, which counting method was used, and the parameter values for the specified subtree.

2.3.2.2 The DARTS Implementation of Halstead's Metrics

The following discussion defines the identification of operators and operands and a method for counting their occurrences in a software design represented as a DARTS tree, according to the generalized technique developed in Section 2.2.3.2. This technique evolved from prior work [Szu 80] and [Szu 81]. The DARTS Halstead Metric capability was implemented in three evolutionary forms, Simple, Uninterpreted, and Interpreted. For this project, only the Interpreted form is discussed.

Since a software design in DARTS can be either abstract or detailed, the technique used to identify operators and operands assumes that a DARTS tree has both uninterpreted and interpreted node forms. Uninterpreted nodes are those which do not contain any primitive operators such as arithmetical or logical operators in the tab. Generally, nodes appearing near the root node of a process architecture tree are more likely to represent uninterpreted functions. These nodes are generally identified as unique operators and contribute a count of one to each of the basic operator equations.

Interpreted nodes are those which contain primitives such as arithmetic or logical operators in the tab field. Control qualifiers (e.g., if, then, else, etc.) are also considered primitives, as are semicolons used for punctuation. A list of all primitive operators which are interpreted by the DARTS Halstead module is shown in Table 7.

Table 7. DARTS Software Science Operator Primitives.

sin	if	and	dur	|	¬=	¬>	random
cos	then	or	lmt	*	<	/	negexp
tan	else	eq	loc	||	>	/*	poisson
arcsin	call	ne	var	&	<>	*/	normal
arccos	while	ge	bgn	¬	<=	,	blk
arctan	until	gt	end	**	>=	;	
log	repeat	le	mod	-	=<	(	
log2	case of	lt	inv	+	=>	)	
log10	print		seg	=	¬<		

The Halstead operands in a DARTS design are those data items appearing in the tab field which are not operators. The Potential Volume (V*) is determined by evaluating η_2^* as the number of data items on the INDATA (input data), and OUTDATA (output data) lists of the top node of the tree under measurement.

2.3.2.2.1 SPECIAL PROCESSING: DARTS uses a hierarchical representation technique which allows a user to truncate trees at any level. This is useful when it is advantageous to hide some details of a design to show only the higher levels. Truncation often leaves ambiguous decision nodes (i.e., decision nodes with no offspring). When this occurs, the node is redefined to be a unique uninterpreted functional node.

2.4 USING THE DARTS DESIGN METRICS

Preceding sections of this report have identified metrics which purport to determine the quality of software designs. These metrics can, at this time, be used as comparators between functionally equivalent but different designs but not as yardstick measures on designs in isolation. Several articles in the literature have included examples of good programming style contrasted with implementations which lack proper organization, structure, and clarity. In some cases, the examples already contain relevant metric data, though in general, the data is derived manually from the code. This section is an attempt to provide some empirical data to support the utility of automated design-aids and metrics by applying the metrics to designs encoded in the DARTS database.

In the two sections to follow, both the Halstead and McCabe metrics, as implemented in DARTS, are used to evaluate some simple and complex designs which are encoded in the DARTS data-base.

2.4.1 Simple Examples

In this section, two examples have been taken from the literature [Ker 74]. These examples are part of the CACM collection, "programming style", containing examples and counterexamples. They were carefully chosen by the authors to depict obvious differences between good and bad code. The examples chosen were also evaluated by the Halstead metric in an article by Gordon [Gor 79]. The results of DARTS Halstead analysis and Gordon's are compared. In addition, the McCabe Cyclomatic Complexity is also calculated.

2.4.1.1 CACM Example 14a

This simple PL/I billing program was translated into a DARTS detailed design, then analyzed by the DARTS Halstead and McCabe metrics. The DARTS representation of the program can be found in Section B.1 of Appendix B. Figure 6 shows the PL/I code and Gordon's [Gor 79] Halstead metric analysis. Table 8 shows the DARTS Halstead and McCabe metric analysis printout tables for comparison.

```
IF QTY > 10
     THEN IF QTY > 200
          THEN IF QTY > =500
               THEN BILL A=BILL A + 1.00;
               ELSE BILL A=BILL A + 0.50;
          ELSE;
     ELSE BILL A=0.00.
```

The code of Example 14A.

Number	η_1	η_2	N_1	N_2	V	$1/\hat{L}$	$\hat{E}$
14A	8	8	21	14	140	7.00	980

Figure 6. PL/I Code and Gordon's Metric Data - CACM 14a

2.4.1.2 CACM Example 14b

This simple PL/I billing program was translated into a DARTS detailed design, then analyzed by the DARTS metrics. The DARTS representation of the program can be found in Section B.2 of Appendix B. Figure 7 shows the PL/I code and Gordon's [Gor 79] Halstead metric analysis. Table 9 shows the DARTS Halstead and McCabe metric analysis printout tables for comparison.

Table 8. DARTS Halstead and McCabe Analysis - CACM 14a

```
CSDL  ***  DESIGN-AIDS          TOPNODE ID:3           PAGE  1
FOR REAL-TIME SYSTEMS           ALL GENERATIONS        DATE: 08 SEPT 1983
HALSTEAD METRIC                 DATABASE IS:  T15A     TIME: 14:37:20
COUNTING METHOD: INTERPRETED    USER IS: AJR1392
```

OPERATORS	COUNT	OPERANDS	COUNT
IF	3	QTY	3
GT	2	10	1
THEN	3	200	1
GE	1	500	1
EQ	3	BILLA	5
+	2	1.00	1
;	4	0.50	1
ELSE	3	0.00	1
8	21	8	14

DISTINCT OPERATORS	8
DISTINCT OPERANDS	8
TOTAL OPERATORS	21
TOTAL OPERANDS	14
VOCABULARY	16
DESIGN LENGTH	35
ESTIMATED LENGTH	48.0
PERCENT OFF	-37.14
DESIGN VOLUME	140.000
POTENTIAL VOLUME	4.755
DESIGN LEVEL	0.034
ESTIMATED DESIGN LEVEL	0.143
INTELLIGENCE CONTENT	20.000
LANGUAGE LEVEL	0.161
ESTIMATED LANGUAGE LEVEL	2.857
EFFORT	4122.074
ESTIMATED EFFORT	980.000

```
CSDL  ***  DESIGN-AIDS          TOPNODE ID:3           PAGE  3
FOR REAL-TIME SYSTEMS           ALL GENERATIONS        DATE: 23 AUG 1983
MCCABE METRIC                   DATABASE IS:  T15A     TIME: 13:40:20
                                USER IS: AJR1392
```

MODULE NAME	#DECISIONS	#PREDICATES	MCCABE INTERVAL LOWER BOUND	MCCABE INTERVAL UPPER BOUND
3	3	3	4	4
TOTAL FOR SUBSYSTEM	3	3	4	4

```
IF QTY >=500
     THEN BILL A=BILL A + 1.00;
     ELSE IF QTY > 200
          THEN BILL A=BILL A + 0.50;
          ELSE IF QTY <=10
               THEN BILL A=0.00;
```

The code of Example 14B.

Number	η_1	η_2	N_1	N_2	V	$1/\hat{L}$	$\hat{E}$
14B	9	8	19	14	135	7.88	1062

Figure 7. PL/I Code and Gordon's Metric Data - CACM 14b

2.4.1.3 CACM Example 15a

This simple PL/I program was translated into a DARTS detailed design, then analyzed by the DARTS metrics. The DARTS representation of the program can be found in Section B.3 of Appendix B. Figure 8 shows the PL/I code and Gordon's [Gor 79] Halstead metric analysis. Table 10 shows the DARTS Halstead and McCabe metric analysis printout tables for comparison.

Table 9. DARTS Halstead and McCabe Analysis - CACM 14b

CSDL ** DESIGN-AIDS	TOPNODE ID:4	PAGE 1
FOR REAL-TIME SYSTEMS	ALL GENERATIONS	DATE: 30 MAR 1983
HALSTEAD METRIC	DATABASE IS: T15A	TIME: 16:40:48
COUNTING METHOD: INTERPRETED	USER IS: AJR1392	

OPERATORS	COUNT	OPERANDS	COUNT
IF	3	QTY	3
GE	1	500	1
THEN	3	BILLA	5
EQ	3	1.00	1
+	2	200	1
;	3	0.50	1
ELSE	2	10	1
GT	1	0.00	1
LE	1		
9	19	8	14

DISTINCT OPERATORS	9
DISTINCT OPERANDS	8
TOTAL OPERATORS	19
TOTAL OPERANDS	14
VOCABULARY	17
DESIGN LENGTH	33
ESTIMATED LENGTH	52.5
PERCENT OFF	-59.18
DESIGN VOLUME	134.886
POTENTIAL VOLUME	4.755
DESIGN LEVEL	0.035
ESTIMATED DESIGN LEVEL	0.127
INTELLIGENCE CONTENT	17.128
LANGUAGE LEVEL	0.168
ESTIMATED EFFORT	1062.229
EFFORT	3826.446

CSDL ** DESIGN-AIDS	TOPNODE ID:4	PAGE 4
FOR REAL-TIME SYSTEMS	ALL GENERATIONS	DATE: 23 AUG 1983
MCCABE METRIC	DATABASE IS: T15A	TIME: 13:40:40
	USER IS: AJR1392	

MODULE NAME	#DECISIONS	#PREDICATES	MCCABE INTERVAL LOWER BOUND	MCCABE INTERVAL UPPER BOUND
4	3	3	4	4
TOTAL FOR SUBSYSTEM	3	3	4	4

```
IF X>=Y
      THEN IF Y>=Z;
            THEN SMALL = Z;
            ELSE SMALL = Y;
ELSE IF X>=Z
            THEN SMALL = Z;
            ELSE SMALL = X;
```

The code of Example 15A

Number	η_1	η_2	N_1	N_2	V	$1/\hat{L}$	$\hat{E}$
15A	6	4	20	14	113	10.50	1186

Figure 8. PL/I Code and Gordon's Metric Data - CACM 15a

2.4.1.4 CACM Example 15b

This simple PL/I program was translated into a DARTS detailed design, then analyzed by the DARTS metrics. The DARTS representation of the program can be found in Section B.4 of Appendix B. Figure 9 shows the PL/I code and Gordon's [Gor 79] Halstead metric analysis. Table 11 shows the DARTS Halstead and McCabe metric analysis printout tables for comparison.

Table 10. DARTS Halstead and McCabe Analysis - CACM 15a

CSDL ** DESIGN-AIDS	TOPNODE ID:2	PAGE 1
FOR REAL-TIME SYSTEMS	ALL GENERATIONS	DATE: 08 SEPT 1983
HALSTEAD METRIC	DATABASE IS: T15A	TIME: 14:36:26
COUNTING METHOD: INTERPRETED	USER IS: AJR1392	

OPERATORS	COUNT	OPERANDS	COUNT
IF	3	X	3
GE	3	Y	3
THEN	3	Z	4
EQ	4	SMALL	4
;	4		
ELSE	3		
6	20	4	14

DISTINCT OPERATORS	6
DISTINCT OPERANDS	4
TOTAL OPERATORS	20
TOTAL OPERANDS	14
VOCABULARY	10
DESIGN LENGTH	34
ESTIMATED LENGTH	23.5
PERCENT OFF	30.85
DESIGN VOLUME	112.946
POTENTIAL VOLUME	2.000
DESIGN LEVEL	0.018
ESTIMATED DESIGN LEVEL	0.095
INTELLIGENCE CONTENT	10.757
LANGUAGE LEVEL	0.035
ESTIMATED LANGUAGE LEVEL	1.024
EFFORT	6378.348
ESTIMATED EFFORT	1185.928

CSDL ** DESIGN-AIDS	TOPNODE ID:2	PAGE 1
FOR REAL-TIME SYSTEMS	ALL GENERATIONS	DATE: 23 AUG 1983
MCCABE METRIC	DATABASE IS: T15A	TIME: 13:40:01
	USER IS: AJR1392	

			MCCABE INTERVAL	
MODULE NAME	#DECISIONS	#PREDICATES	LOWER BOUND	UPPER BOUND
IS X LARGER THAN OR EQUAL TO Y	3	3	4	4
TOTAL FOR SUBSYSTEM IS X LARGER THAN OR EQUAL TO Y	3	3	4	4

```
SMALL = X;
IF Y < SMALL
      THEN SMALL = Y;
IF Z < SMALL
      THEN SMALL = Z;
```

The code of Example 15B.

Number	η_1	η_2	N_1	N_2	V	$1/\hat{L}$	$\hat{E}$
15B	5	4	12	10	70	6.25	436

Figure 9. PL/I Code and Gordon's Metric Data - CACM 15b

2.4.1.5 *Analysis of the Simple Examples Experiment*

The subjective evaluation by the authors of the programs [Ker 74] suggests that Example 14a is better than Example 14b, and Example 15b is better than Example 15a. These findings were reinforced by Gordon's manual application of the Halstead metrics [Gor 79]. The automated DARTS Halstead analysis of these same examples reproduced Gordon's data, and, in addition, the DARTS McCabe metric verified the results in one of the two cases. In the case of Example 14, the metric values were the same. By examining the control structure of these programs, it is obvious that the McCabe metric cannot distinguish between these two similar designs at the level of detail presented.

2.4.2 Complex Examples

In this section, the metrics described in Section 2.3 are applied to example real-time system designs. In order to illustrate the ability of the metrics to distinguish between designs of differing quality, two candidate design solutions to a complex problem are introduced. The designs are expressed using the automated design medium of DARTS, and the metrics are applied automatically.

Table 11. DARTS Halstead and McCabe Analysis - CACM 15b

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS HALSTEAD METRIC COUNTING METHOD: INTERPRETED	TOPNODE ID:8 ALL GENERATIONS DATABASE IS: T15A USER IS: AJR1392	PAGE 1 DATE: 08 SEPT 1983 TIME: 14:36:52

OPERATORS	COUNT	OPERANDS	COUNT
EQ ; IF LT THEN	3 3 2 2 2	SMALL X Y Z	5 1 2 2
5	12	4	10

DISTINCT OPERATORS	5
DISTINCT OPERANDS	4
TOTAL OPERATORS	12
TOTAL OPERANDS	10
VOCABULARY	9
DESIGN LENGTH	22
ESTIMATED LENGTH	19.6
PERCENT OFF	10.87
DESIGN VOLUME	69.738
POTENTIAL VOLUME	2.000
DESIGN LEVEL	0.029
ESTIMATED DESIGN LEVEL	0.160
INTELLIGENCE CONTENT	11.158
LANGUAGE LEVEL	0.057
ESTIMATED LANGUAGE LEVEL	1.785
EFFORT	2431.719
ESTIMATED EFFORT	435.865

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS MCCABE METRIC	TOPNODE ID:8 ALL GENERATIONS DATABASE IS: T15A USER IS: AJR1392	PAGE 2 DATE: 23 AUG 1983 TIME: 13:40:11

			MCCABE INTERVAL	
MODULE NAME	#DECISIONS	#PREDICATES	LOWER BOUND	UPPER BOUND
FIND THE LEAST OF THREE VALUES	2	2	3	3
TOTAL FOR SUBSYSTEM FIND THE LEAST OF THREE VALUES	2	2	3	3

The following sections introduce the sample problem, present the two candidate designs, and apply the metrics to the designs. The section closes with a summary.

2.4.2.1 The Experiment Controller Example

The example chosen is an experiment controller, first described by Mendelbaum and Madaule [Men 75], and later discussed by Chow [Cho 78]. The computer controls a series of laboratory experiments, positioning a burette piston prior to each experiment, and then records and analyzes sensor data (e.g., determining solute concentration). A report is printed at the end of a series of experiments; however, the user has a switch which causes the report to be issued at intermediate points if desired. The detailed requirements for this system are presented in Table 12, and Figure 10 is a pictorial representation of the system.

This example was selected for a variety of reasons.

- It illustrates a number of real-time design issues, including asynchronous user interraction, and timer-driven cyclic behavior.
- There are two designs derived from the same requirements which can be compared both subjectively and by the metrics.
- It is a complex, yet simply illustrated, example.

Each design is depicted using DARTS. Figure 11 shows the first version of the design, labeled Design 1, and Figure 12 shows the second version, labeled Design 2. The DARTS metric analysis of these designs is discussed in the next section.

2.4.2.2 Metric Analysis of the Experiment Controller Designs

In this section, the results of the DARTS Halstead and McCabe metric analysis of the Experiment Controller Example designs are presented and discussed.

Design 1, as depicted in Figure 11, was analyzed using DARTS. Table 13 shows 1) the raw counts of operators and operands used to calculate the Halstead metrics, 2) the Halstead metrics, and 3) the McCabe Cyclomatic Complexity Interval.

Design 2, as depicted in Figure 12, was also analyzed using DARTS. Table 14 shows 1) the raw counts of operators and operands used to calculate the Halstead metrics, 2) the Halstead metrics, and 3) the McCabe Cyclomatic Complexity Interval.

Table 12. Requirements for the Experiment Controller

1. A computer system is needed to control a series of laboratory experiments (see Figure 4-1).

2. A burette step motor compresses a burette piston. Each step of the motor corresponds to a given poured volume. Following each step an interrupt signal i_B is sent to the computer.

3. An electric cell sensor enables the computer to measure concentrations in a tank.

4. Experimental data are sent to the user via a printer. When the printer has finished printing a list of data, an interrupt signal i_P is sent to the computer.

5. A user switch enables the user to interrupt and obtain a status report during the experiments by emitting a signal i_U.

6. The computer system can make use of a timer to request an interrupt signal i_T after a given fixed interval.

7. Prior to performing the first experiment, an INITIALIZATION task is performed, followed by an initial reading of the cell sensor and the MEASURE task.

8. The computer has access to a count of the number of experiments and an experiment table containing instructions for each experiment (the initial definition of this table is not part of the problem).

9. For each experiment, the burette controller motor is operated for a number of steps (as determined from the experiment table).

10. Next a series of measurements is performed at fixed time intervals (as determined from the experiment table), by starting the timer, reading the sensor when the timer interrupt occurs, and performing the MEASURE task. The series is terminated after a fixed number of measurements (as determined from the experiment table).

11. After each measurement, if the user has sent the signal i_U, the COMPUTE and LIST tasks are performed, and a status report is sent to the printer.

12. After each experiment, it is determined whether the series of experiments is finished. If not, steps 9 through 11 above are repeated. If so, the COMPUTE and LIST tasks are performed, and the final report is sent to the printer.

13. The report cannot be sent to the printer if the printer is already in use (it is delayed until the printer is available).

14. While a user print request is in execution, additional user print requests are ignored.

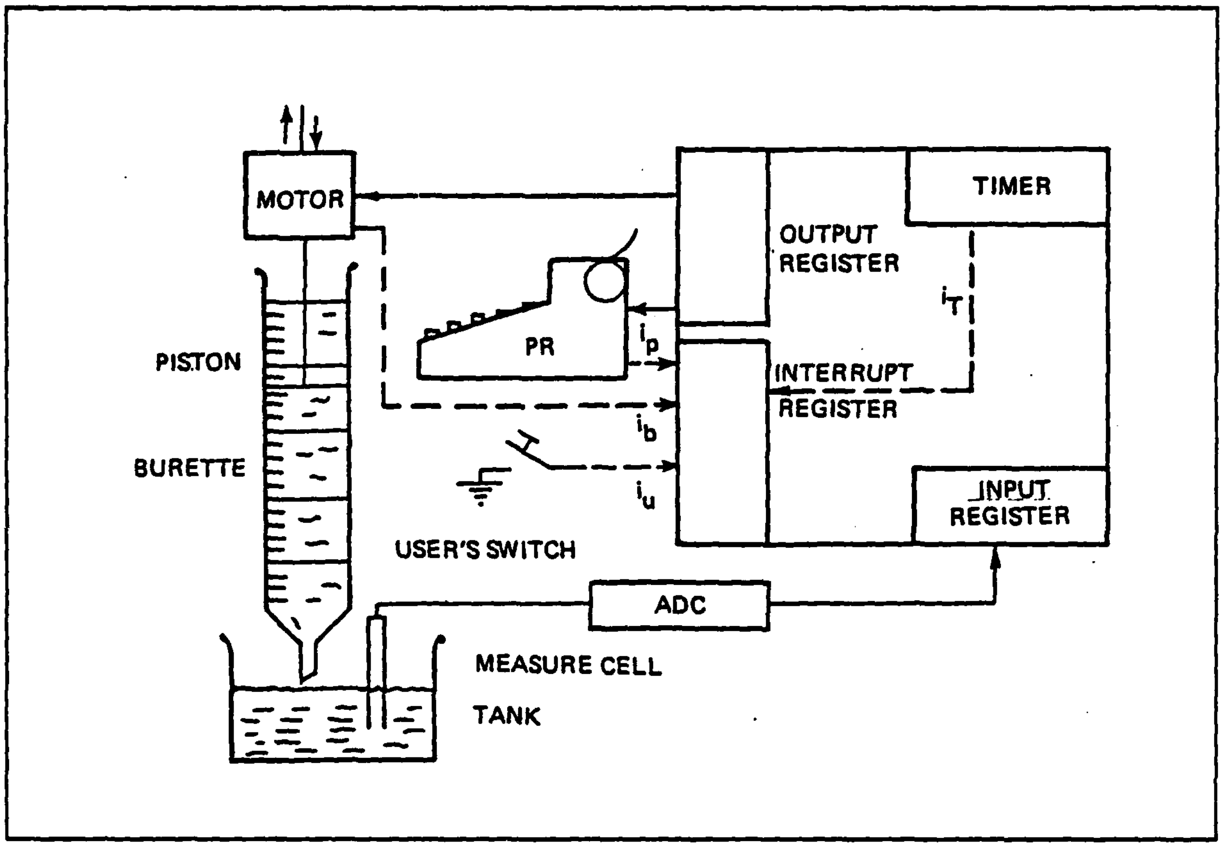


Figure 10. Experiment Controller System

For both of these designs, the minimum number of unique operands is 11. This is derived from counting the input and output data items at the top of each design tree. This information is obtained from the data-flow tables for each of the designs, which appear in Appendix E. As Table 13 and Table 14 show, the important complexity metrics of Halstead's theory (length, volume, and effort), and McCabe's Cyclomatic Complexity suggest that Design 1 is less complex than Design 2. This agrees with a subjective assessment that a proficient designer might make in comparing these designs.

2.5 *DESIGN METRICS AND ADA*

This section considers how Ada may be used as a design representation medium during the design phase, and how the Halstead metrics may be measured

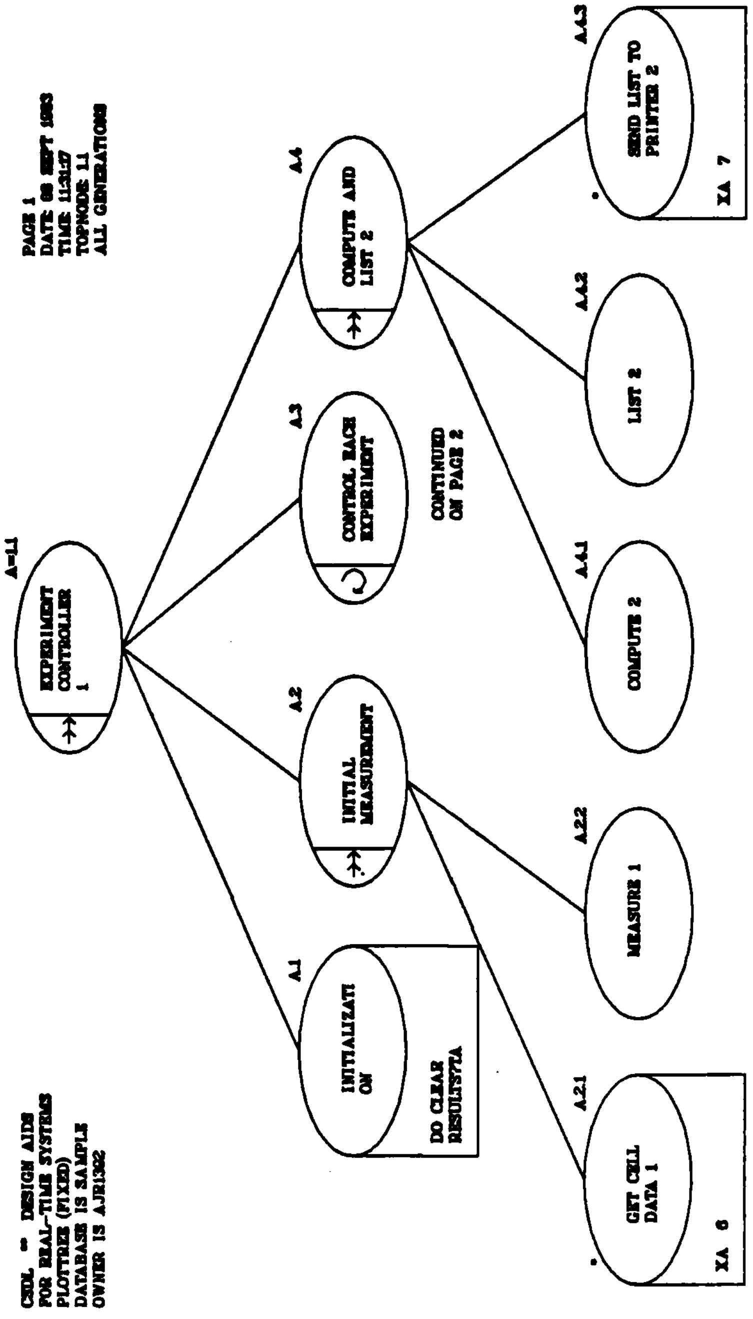


Figure 11. Experiment Controller - Design 1 (Part 1 of 3)

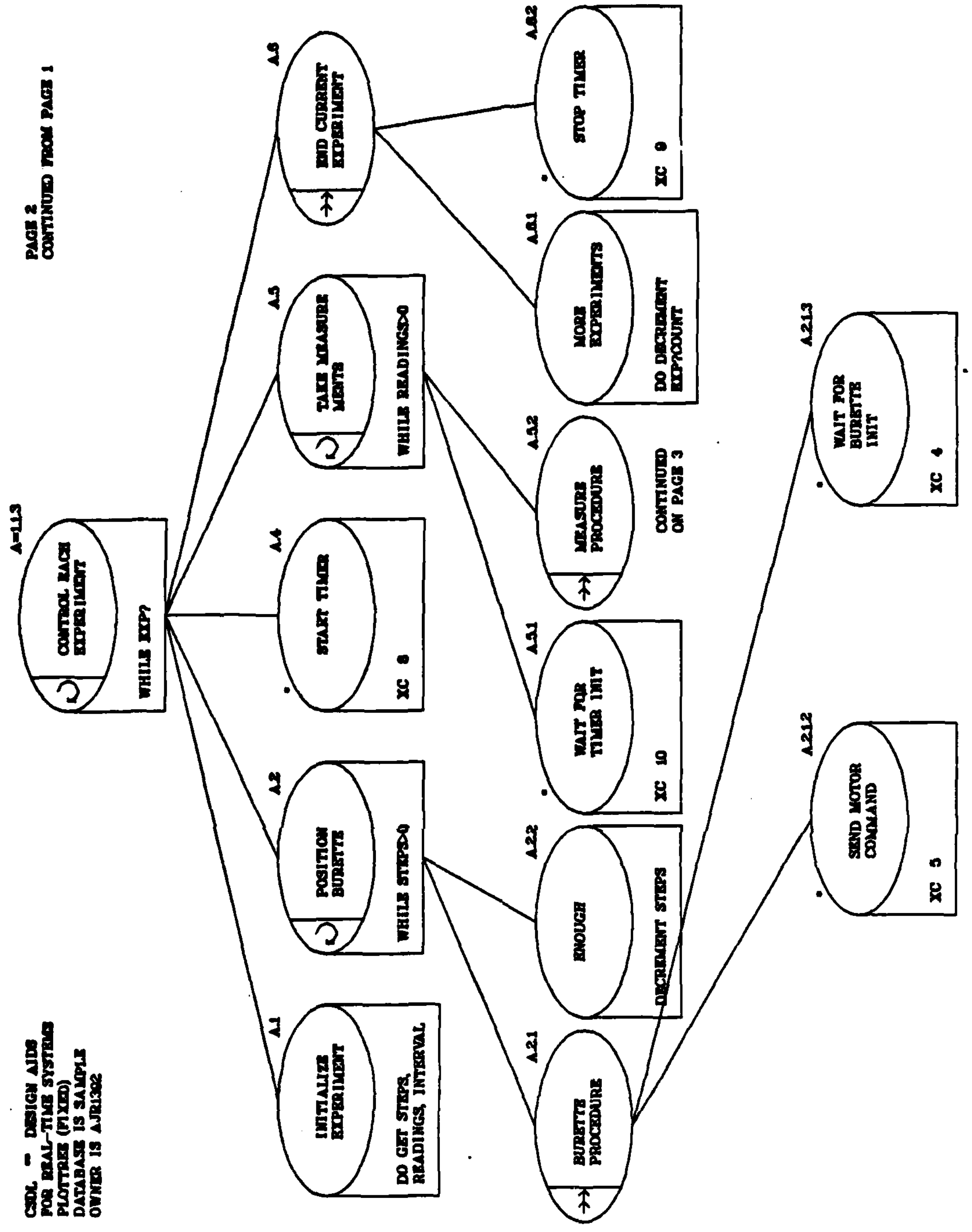


Figure 11. Experiment Controller - Design 1 (Part 2 of 3)

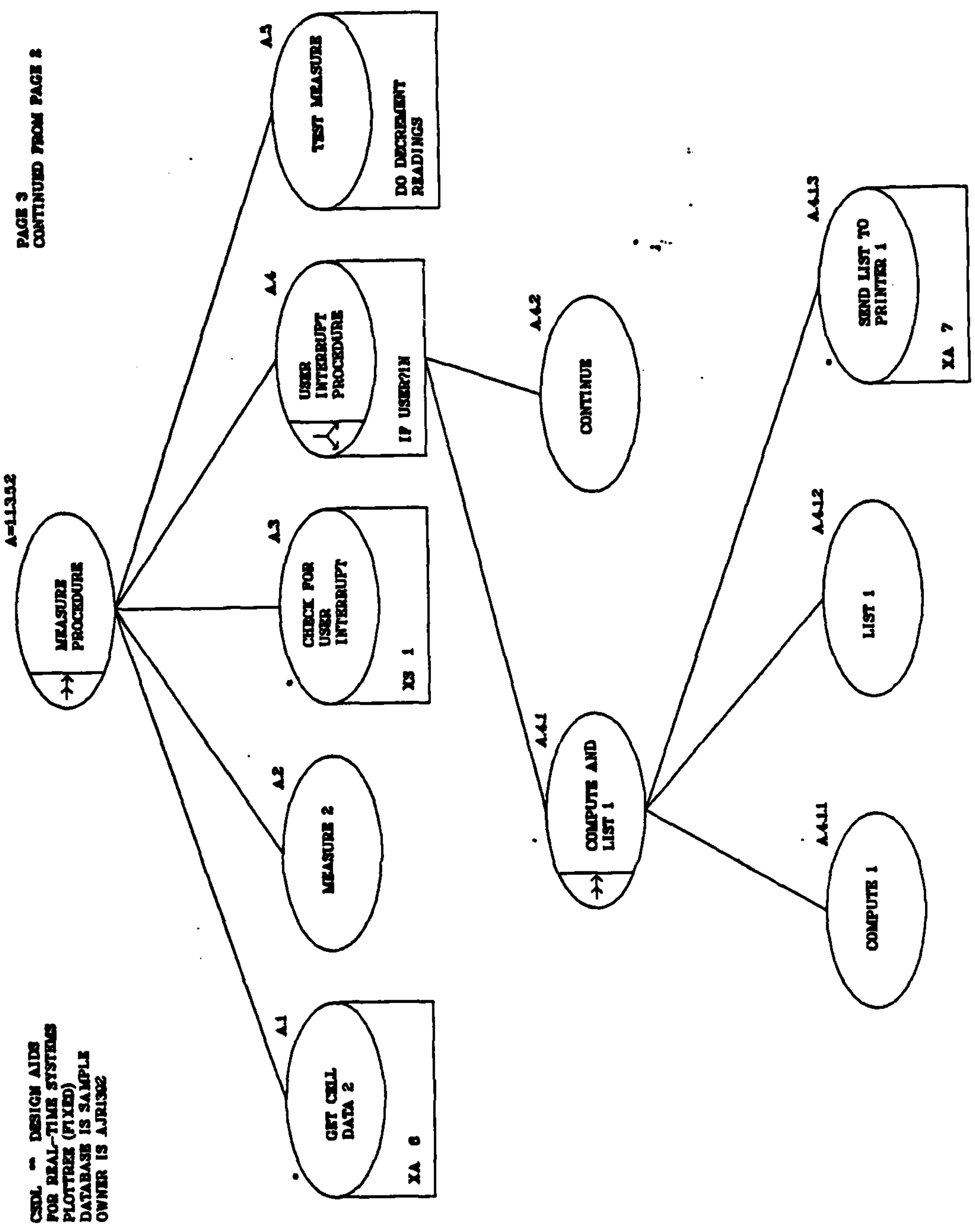


Figure 11. Experiment Controller - Design 1 (Part 3 of 3)

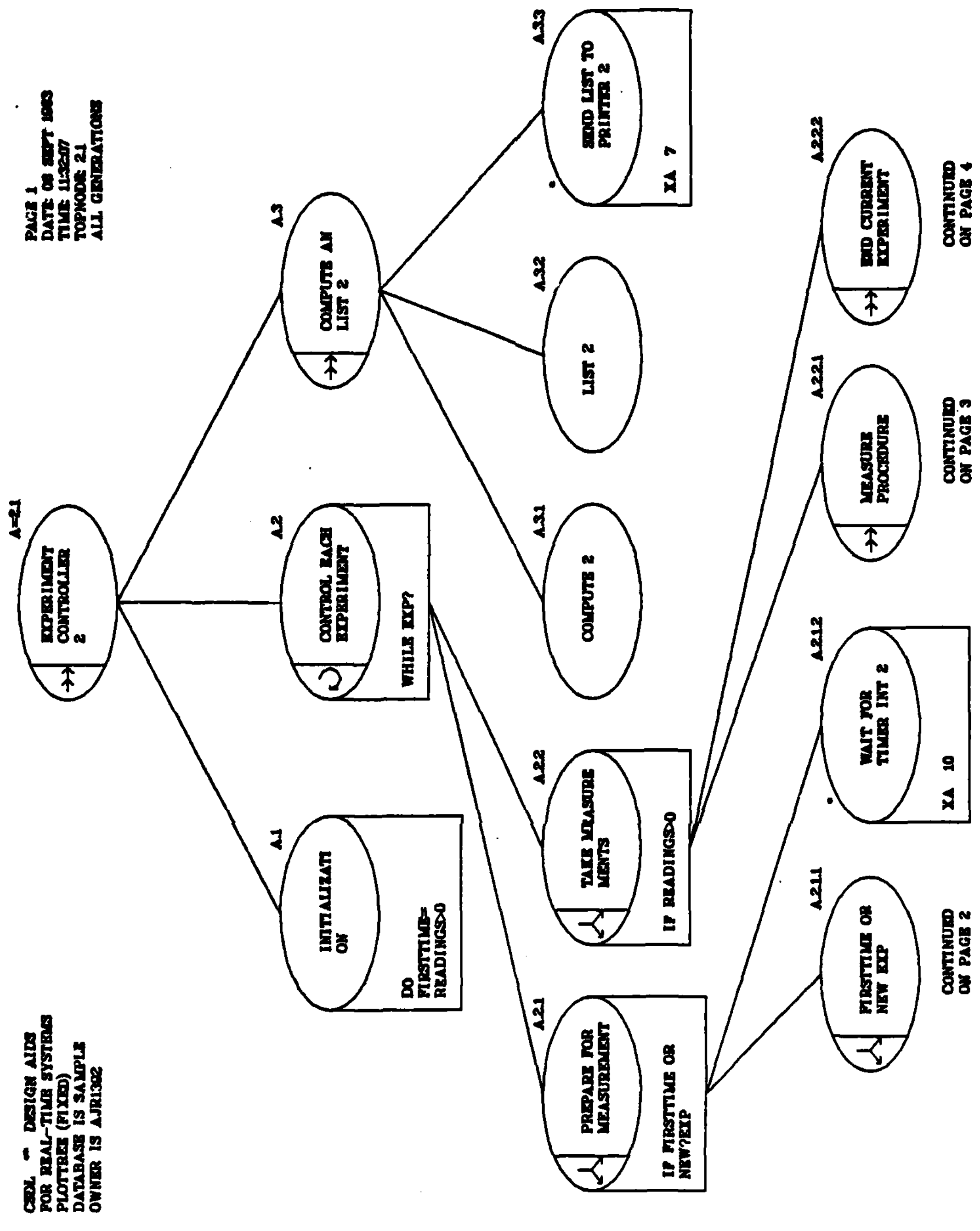


Figure 12. Experiment Controller - Design 2 (Part 1 of 5)

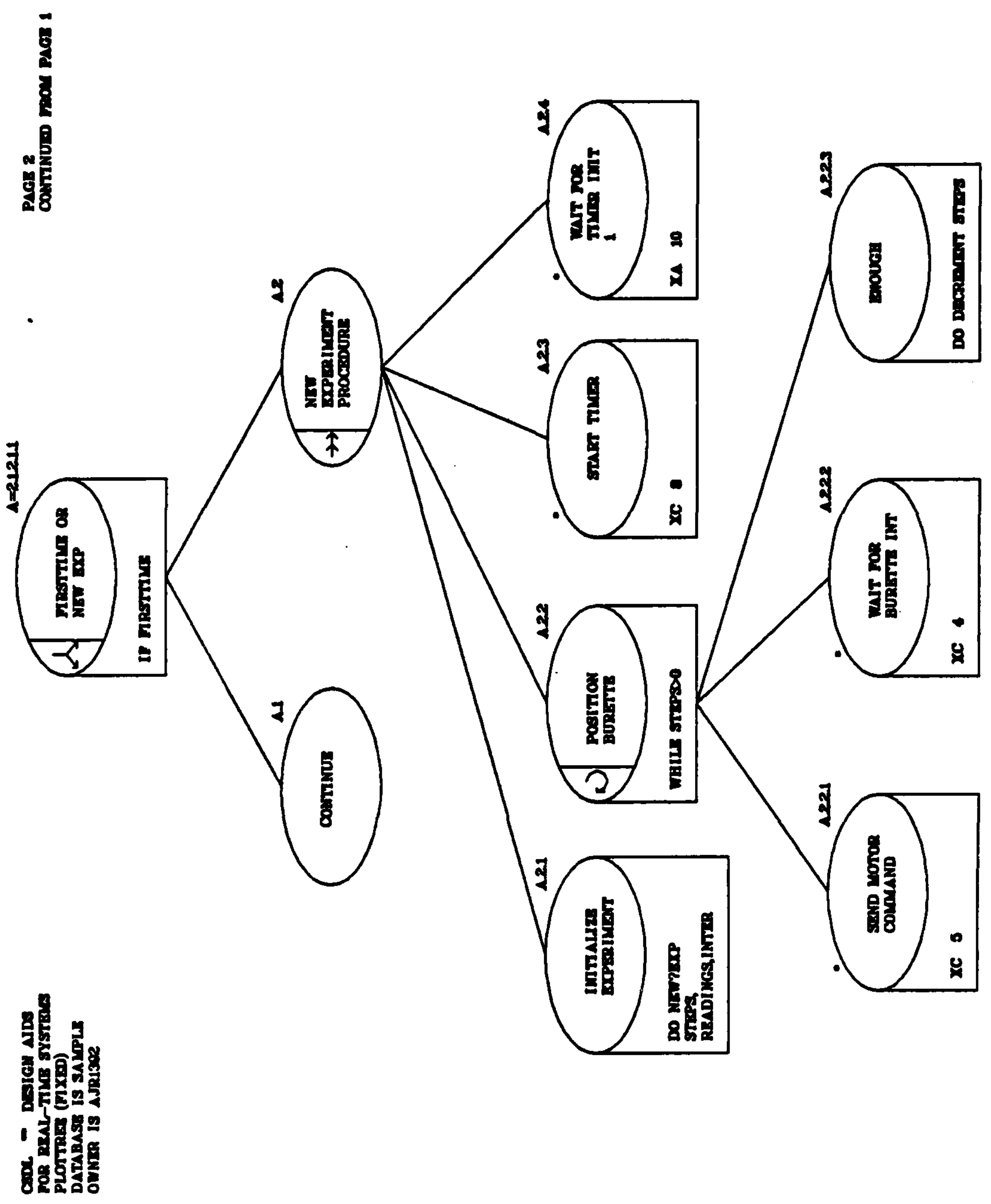


Figure 12. Experiment Controller - Design 2 (Part 2 of 5)

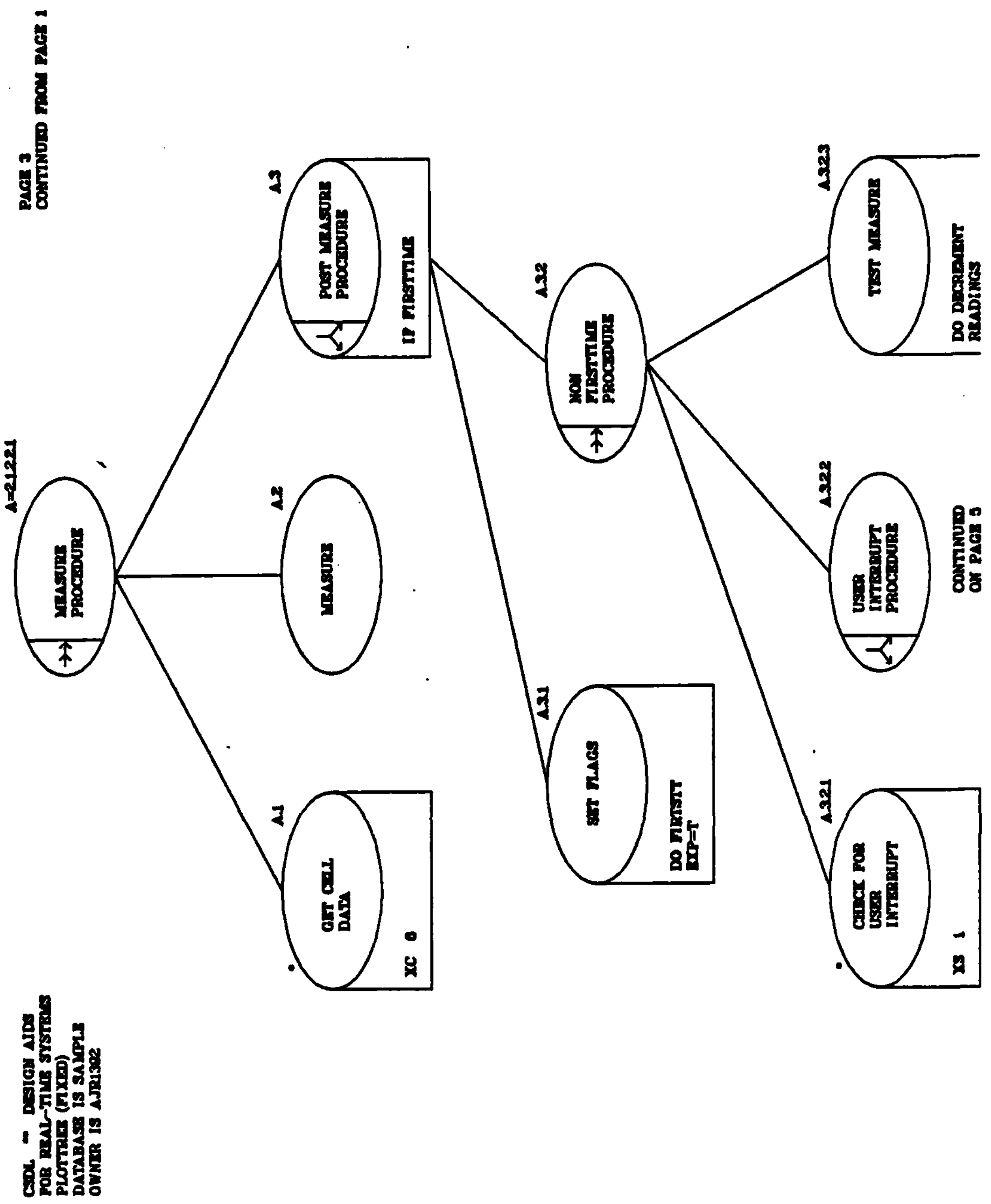


Figure 12. Experiment Controller - Design 2 (Part 3 of 5)

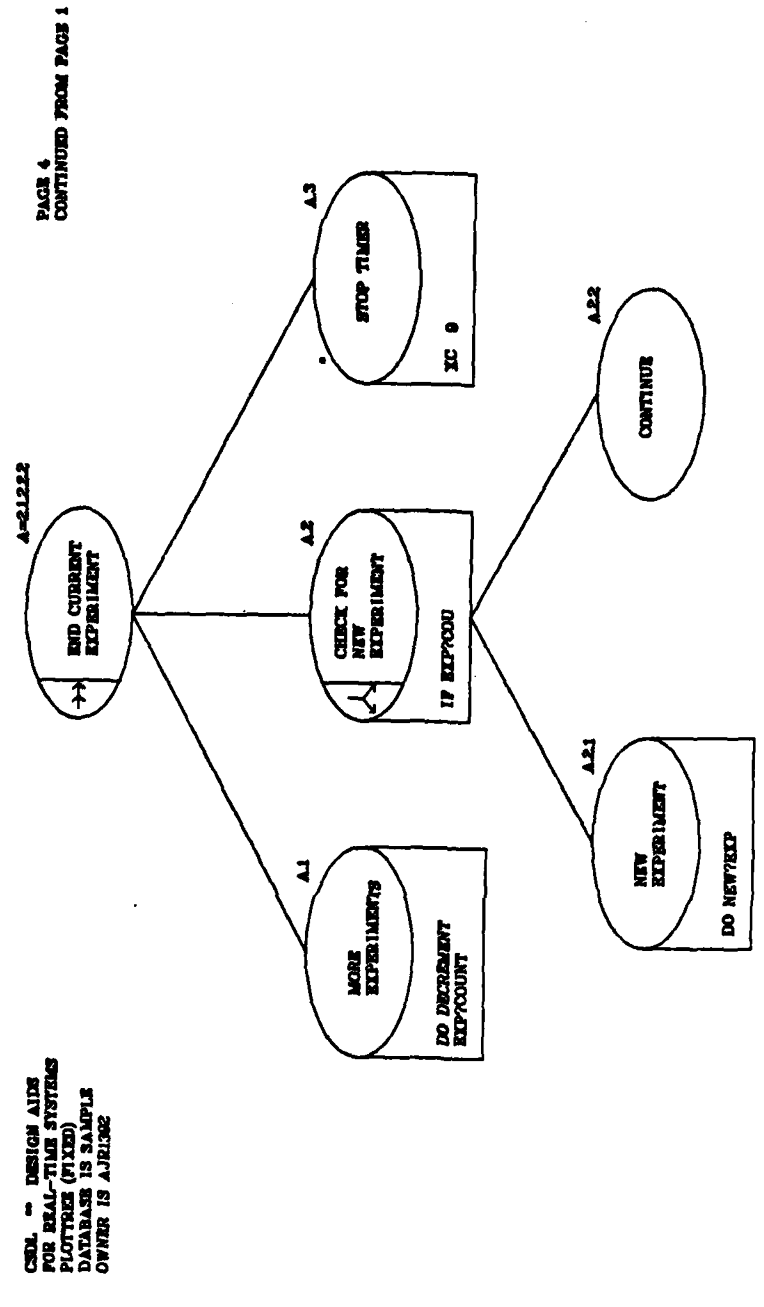


Figure 12. Experiment Controller - Design 2 (Part 4 of 5)

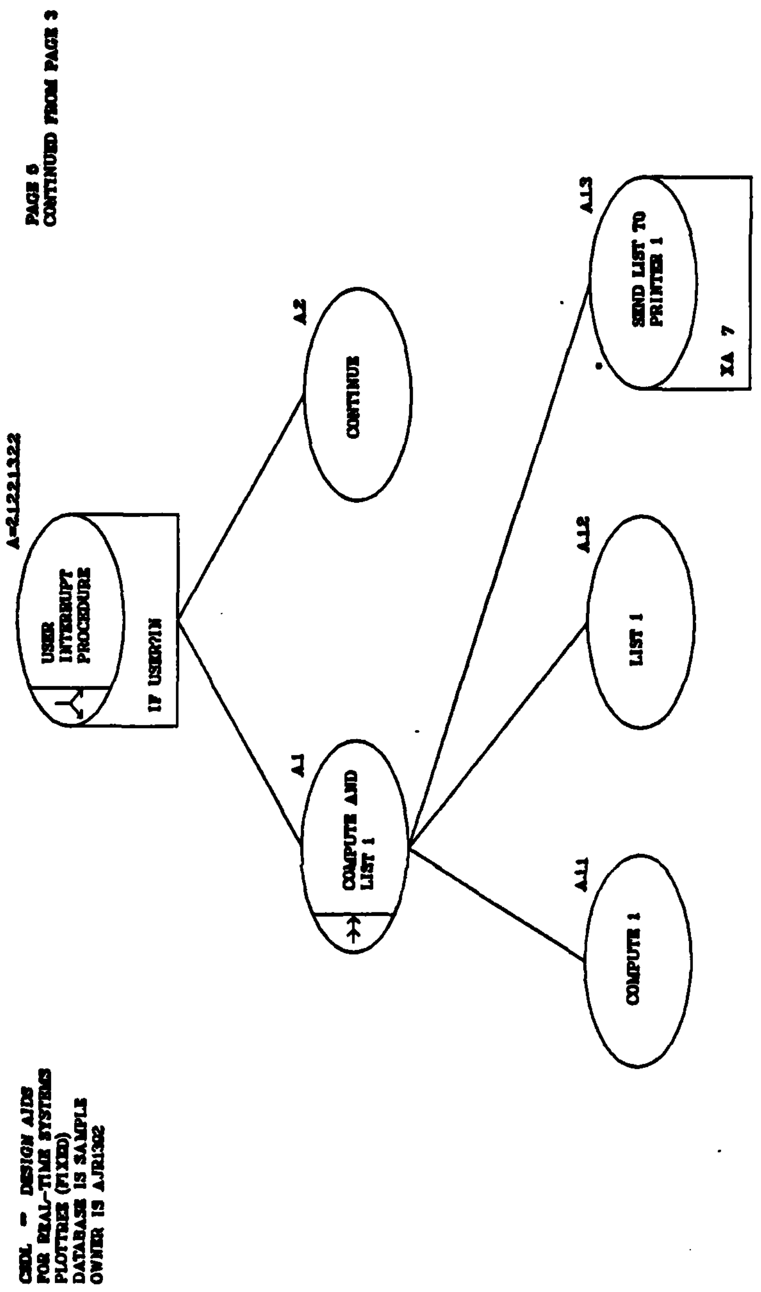


Figure 12. Experiment Controller - Design 2 (Part 5 of 5)

Table 13. DARTS Metrics Summary - Design 1

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS HALSTEAD METRIC COUNTING METHOD: INTERPRETED	TOPNODE ID:1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 1 DATE: 24 AUG 1983 TIME: 19:46:58

OPERATORS	COUNT	OPERANDS	COUNT
WHILE	3	DO	4
,	2	CLEAR	1
IF	1	RESULTS_TABLE	1
		XA	4
		6	2
		EXP_COUNT	2
		0	6
		GET	1
		STEPS	3
		READINGS	3
		INTERVAL	1
		XC	5
		5	1
		4	1
		DECREMENT	3
		8	1
		10	1
		XS	1
		1	1
		USER_INTERRUPT	1
		7	2
		9	1
3	6	22	46

DISTINCT OPERATORS	3
DISTINCT OPERANDS	22
TOTAL OPERATORS	6
TOTAL OPERANDS	46
VOCABULARY	25
DESIGN LENGTH	52
ESTIMATED LENGTH	102.9
PERCENT OFF	-97.81
DESIGN VOLUME	241.481
POTENTIAL VOLUME	2.000
DESIGN LEVEL	0.008
ESTIMATED DESIGN LEVEL	0.319
INTELLIGENCE CONTENT	76.994
LANGUAGE LEVEL	0.017
ESTIMATED LANGUAGE LEVEL	24.549
EFFORT	29156.445
ESTIMATED EFFORT	757.371

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS MCCABE METRIC	TOPNODE ID:1.1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 1 DATE: 10 AUG 1983 TIME: 20:22:20

			MCCABE INTERVAL	
MODULE NAME	#DECISIONS	#PREDICATES	LOWER BOUND	UPPER BOUND
EXPERIMENT CONTROLLER 1	4	4	5	5
TOTAL FOR SUBSYSTEM EXPERIMENT CONTROLLER 1	4	4	5	5

Table 14. DARTS Metrics Summary - Design 2

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS HALSTEAD METRIC COUNTING METHOD: INTERPRETED	TOPNODE ID:S ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 1 DATE: 24 AUG 1983 TIME: 19:47:30

OPERATORS	COUNT	OPERANDS	COUNT
;	3	DO	7
WHILE	2	FIRSTTIME	5
IF	6	N	1
OR	1	NEW_EXP	5
,	2	T	6
		READINGS	4
		0	10
		EXP_COUNT	3
		F	4
		GET	1
		STEPS	3
		INTER	1
		XC	5
		5	1
		4	1
		DECREMENT	3
		8	1
		XA	4
		10	2
		6	1
		XB	1
		1	1
		USER_INTERRUPT	1
		7	2
		9	1
5	14	25	74

DISTINCT OPERATORS	5
DISTINCT OPERANDS	25
TOTAL OPERATORS	14
TOTAL OPERANDS	74
VOCABULARY	30
DESIGN LENGTH	88
ESTIMATED LENGTH	127.7
PERCENT OFF	-45.12
DESIGN VOLUME	431.806
POTENTIAL VOLUME	2.000
DESIGN LEVEL	0.005
ESTIMATED DESIGN LEVEL	0.135
INTELLIGENCE CONTENT	58.352
LANGUAGE LEVEL	0.009
ESTIMATED LANGUAGE LEVEL	7.885
EFFORT	93228.438
ESTIMATED EFFORT	3195.368

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS MCCABE METRIC	TOPNODE ID:S.1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 2 DATE: 10 AUG 1983 TIME: 20:22:35

			MCCABE INTERVAL	
MODULE NAME	#DECISIONS	#PREDICATES	LOWER BOUND	UPPER BOUND
EXPERIMENT CONTROLLER 2	8	8	9	9
TOTAL FOR SUBSYSTEM EXPERIMENT CONTROLLER 2	8	8	9	9

from such a design. First, to what extent the information necessary to the products of the design phase may be expressed in Ada is examined. Then, a counting method and guidelines for measuring the Halstead metrics from an Ada design are proposed. This method is illustrated with an example, and some of the issues raised are discussed.

2.5.1 Motivation

As Ada compilers and Programming Support Environments are nearing availability for use on actual projects, there is much interest in using Ada, itself, as a design representation medium. This is appealing, because it provides an orderly, evolutionary way to progress from architectural design through detailed design to full implementation. It is appropriate, because the language provides the means for defining objects and operations which are not primitive in Ada, through type, variable, function, procedure, and task declarations; the means for hiding information about the implementation of the objects and operations through packages and private types; and the means for delaying implementation decisions through separate compilation of package specifications and bodies. Thus, a design can be expressed in Ada at any desired level of abstraction; and characteristics such as interface correctness and data typing consistency can be checked through use of the compiler, at any stage of the process.

Objections which have been raised to using Ada as a design representation medium center on the difficulty of restraining designers from premature introduction of implementation detail, and the need, during design, for information which is not best represented in Ada. Subsets and supersets of Ada have been proposed to solve these problems.

2.5.2 The Object-Oriented Design Methodology

To make effective use of Ada as the design representation medium, a design method must be employed which allows easy introduction of detail as design decisions are made, but discourages the premature introduction of detail which might unduly constrain the implementation. One such method is the object-oriented design methodology explicated in [Boo 83]. It is used here to demonstrate the use of Ada as a design medium and is summarized in the following steps ([Boo 83] p. 70):

1. Define the problem.

2. Develop an informal strategy for solving the problem. State the problem solution in English.

3. Formalize the strategy by

a. Identifying the objects used in the solution, and their attributes.

b. Identifying the operations which are performed on the objects during the solution.

c. Establishing the interfaces among the object-operation groups.

d. Implementing the objects and operations.

This last step defines a new problem, so all the steps may be repeated at successively greater levels of detail, until a code-to design is reached.

This method is also consistent with expression of the design using DARTS, since DARTS supports hierarchical decomposition.

2.5.3 Using Ada as a PDL with the Object-Oriented Design Method

Ada's information hiding, data abstraction, and strong typing capabilities make it very suitable for expressing a design which is developed according to the object-oriented design methodology, or another top-down, stepwise refinement method. The objects become constants or variables of appropriate user-defined data types, or tasks; the operations become procedures, functions, task entries, or exceptions; the object-operation groups, which are abstract data types, become packages. The interfaces among the groups are defined by the package specifications and with clauses, and are controlled by the compiler interface and type checking. Ada's capability for separating a subprogram specification and body supports top-down design methods by allowing an abstract data type to be referred to once its interface is specified, before it is actually implemented. In addition, Ada provides means for expressing relationships among objects and operations which are not found in many design languages. These include the ability to specify parallel processing, asynchronous data exchange, interrupts, exceptions, and machine-dependent items.

Ada by itself, however, is not ideal for representing all of the information which is needed during design. Table 15 and Table 16 indicate how the information in each paragraph of the MIL-STD-SDS ([DoD 82]) design documents can be represented in Ada. The paragraph numbers are taken from the associated Data Item Descriptions (DIDs) ([DoD 82] R-DID-110 and R-DID-111).

As these tables show, much of the information called for in the documents can only be represented in Ada as comments. In general, this information falls into two classes: that which describes constraints or properties of the software which are not directly reflected in its textual content, such as execution time, memory size, or data rates; and that which requires reference to

Table 15. MIL-STD SDS Top Level Design Document Information in Ada.

Paragraph number	Means of representation in Ada
1.1	Comments, subprogram name.
1.2	Comments.
2.1	Comments.
2.2	Comments.
3	Comments.
3.1	Package specifications and comments.
3.1.1	Comments, or by describing the hardware with a package specification.
3.1.2	Package specification with comments for data rates.
3.2	Comments.
3.3.1	Task entry specifications and bodies, exception specifications, handlers, representation specifications.
3.3.2	Comments, real-time control statements (e.g. DELAY), PRIORITY pragma.
3.3.3	Package and representation specifications.
3.3.4	Comments, real-time statements.
3.4	Subprogram specification.
3.4.X	Comments.
3.4.X.1	Comments, subprogram specification.
3.4.X.2	Subprogram specification, comments.
3.4.X.3	Subprogram specification, comments, type and interface checking.
3.4.X.4	Exception specifications and handlers, representation specifications, comments.
3.5	Package specification, representation specifications, comments.
3.5.1	Package specifications, type definitions, variable declarations, constraints, representation specifications.
3.5.2	Package specifications, type definitions, variable declarations, constraints, representation specifications, comments.
3.6.all	Package specifications, representation specifications, comments.
4, 5, 7-9	Intentionally left blank in DIDs.
6	Comments.
10	Appendices - not applicable.

Table 16. MIL-STD SDS Detailed Design Document Information in Ada.

Paragraph number	Means of representation in Ada
1.1	Comments.
1.2	Comments, package specification.
2.all	Comments.
3	Comments.
3.1	Subprogram specifications, comments.
3.1.1	Subprogram specifications.
3.1.1.1	Comments, subprogram specifications.
3.1.1.2	Comments.
3.1.1.3	Subprogram specifications, bodies, comments.
3.1.1.3.Y a)	Comments.
3.1.1.3.Y b)	Subprogram specification, declaration part of body.
3.1.1.3.Y c)	Subprogram specification.
3.1.1.3.Y d)	Subprogram specification, WITH and USE clauses, comments.
3.1.1.3.Y e)	WITH and USE clauses, comments.
3.1.1.3.Y f)	Comments.
3.1.1.3.Y g)	Comments, subprogram specification, body.
3.1.1.3.Y h)	Subprogram specification, WITH and USE clauses, subprogram body, comments.
3.1.1.3.Y i)	Comments.
3.2.1.X.6	Task entry specifications and bodies, exception specifications, handlers, representation specifications, comments.
3.2.1.X.7	Representation specifications, comments.
3.2.1.X.8	Comments, real-time statements.
3.2.1.X.9	Package specifications, type definitions, variable declarations, constraints, representation specifications, comments.
3.2.1.X.10	Package specifications, type definitions, variable declarations, constraints, representation specifications, comments.
3.2.1.X.11	Subprogram body.
3.2.1.X.12	Comments.
3.3	Representation specifications, comments.
4, 5, 7-9	Intentionally left blank in DIDs.
6	Comments.
10	Appendices - not applicable.

the project information structure outside the software, such as requirements allocation to components, or references to design analyses. Supersets of Ada, such as Byron ([Gor 83]), teamed with a key concept of the APSE requirements, a unified database for the whole software development process, will do much to alleviate these problems.

Another issue which determines how effective Ada is as a design representation medium is what tools and processing are available for turning the information in the medium into the form which is suitable for the purposes of the user. Since Ada is primarily a programming language, we tend to think of it as input to a compiler. Other tools may use it as input for other purposes, such as generation of documents, flowcharts, variable or module cross-reference lists, data dictionaries, or hierarchical control trees. The utility of Ada as a design medium, then, depends on the analysis capabilities of a compiler and any other tools that may be used for processing the design representation, as well as the expressive power of the language. In particular, a tool to measure metrics might prove useful.

2.5.3.1 Architectural Design

Using the object-oriented design method, the architectural design begins with the expression of the informal problem solution in English. Once the objects, operations, and their interfaces are defined, they may be specified in the top level Ada representation. This usually takes the form of a package specification for each group of objects and operations which is used in the problem solution, and a subprogram specification for the solution itself. Preliminary design ends when enough iterations through the method have been made so that the hierarchy, control, and data interfaces ([Boe 81] p. 48) for all components have been defined to the level of a preexisting software component, or a subprogram which "performs a single well-defined function, can be developed by one person, and is typically 100 to 300 source instructions in size" ([Boe 81] p. 49). As design progresses, the bodies of the packages are filled in with control structures which use new, lower level, objects and operations. These are specified in new packages. The body for the solution subprogram is also filled in to show how the most abstract object-operation groups are used.

The first design problem in [Boo 83] is used, in the following section, as a sample for demonstrating the application of the Halstead metrics. It comprises a subroutine for counting the leaves on a binary tree. The English language statement of the informal problem solution ([Boo 83] p. 72) (the informal architectural design specification) is reproduced in Figure 13 at the end of this section. Figure 14 reproduces the initial package specifications for the counting leaves problem from [Boo 83], pp. 76-77, and Figure 15 reproduces the solution algorithm ([Boo 83] p. 78) which uses these packages as primitives. Together they constitute the architectural design for the problem.

2.5.3.2 Detailed Design

During detailed design, the algorithms for implementing the objects and operations specified in the last level of architectural design are developed in terms of objects and operations which are not visible outside the subprogram which implements them. This may require further type, object and operation definitions, but they are logically local: the problem solution does not use them directly, even though they may be part of a library of commonly available packages. There will tend to be a larger proportion of native Ada constructs and low level utility routines used than in the subprograms developed during architectural design. Here, decisions such as the data structure for local variables, or the order of operations necessary to meet accuracy requirements are made.

Figure 16 shows the full detailed design for the counting leaves problem. The private parts of the initial package specifications have been filled in, and the bodies have been written. The external specification for another package, FIFO_PACKAGE, has been introduced, because it is used in the implementation of the TREE_PACKAGE. It is assumed to be a preexisting package, so its body is not given here. The solution statement has not changed from that given in the architectural design, so it is not repeated.

2.5.4 Using Halstead Metrics on Ada

The Halstead metrics may be applied at several stages during the design development: The informal English *statement* of the problem solution may be measured following the techniques in [Hal 77] Chapter 13 ([Hal 77] pp. 98-110) (this is explained with the example in a later section), and the design expressed in Ada may be measured using the method described in the following section, at any stage of the design or coding phases.

The metrics may be used to provide a way to judge which of several different solution strategies is simpler, based on the objects and operations in which it is stated; to estimate planning parameters from the architectural design; and to provide a reasonableness check on the quality of the design as it evolves. Since the design is expressed in actual Ada code, the same metrics may be applied to the code as it evolves, to monitor complexity as changes are made. The measurements for the architectural design can also be compared with those for the English solution statement to discourage premature implementation decisions.

2.5.4.1 The Counting Method

A design expressed in Ada can be viewed in two ways: as an expression of a problem solution in terms of a language invented for the occasion, comprising the objects and operations needed for the solution; or as an implementation of that solution in the language Ada. In the first case, the objects and operations developed during design are viewed as if they had always existed, and had been part of the designer's vocabulary. The language the designer is speaking is not Ada, but an extremely specific one which solves the problem, but cannot express anything else.

This might be the more useful point of view for comparing competing designs for conciseness as abstract solutions. The second case, however, is a more accurate representation of what actually happens during the design process: the designer invents the objects and operations with which to express the solution. This corresponds to using the Ada constructs which create types, variables, and subprograms. It is proposed here as the point of view more likely to lead to a serviceable method of using the Halstead metrics for comparing designs as practical solutions, and for predicting planning parameters.

Accordingly, in the method proposed below, lexical elements in Ada are designated as operators or operands in accordance with how the they function from the point of view of the person who is writing in the language Ada, giving directions to an abstract machine which will create and manipulate the objects specified. As design progresses, the abstract machine becomes particularized as the real machine.

This point of view has other benefits: It allows all the code available to be included in the measurements at any stage, so complicated provisos are not necessary for different levels of detail. It allows different strategies to be devised for selecting portions of the the design to be measured to tailor the method for a particular use of the metrics, such as predicting planning parameters. It allows the same method to be used throughout the design and coding phases and it is consistent with earlier efforts at measuring the Halstead metrics in high level languages.

2.5.4.1.1 IDENTIFYING OPERATORS AND OPERANDS

With this philosophy, the entities which are operations in the object-oriented design statements are usually Halstead operands of Ada keywords which are Halstead operators. For example, a procedure or task entry call will be a verb phrase in the solution statement. In Ada, its declaration is viewed as a command to create a procedure or entry and make accessible its name and external interface. The operator is the implicit "create" represented by the reserved word **procedure** or **entry** and the operand is the procedure/entry name. The invocation of the procedure or entry is viewed as a "call" or "invoke" operator with the procedure/entry name as its operand. Similarly, the reserved words **type** and **package** are further particularizations of the general "create" operand, with the type and package names as operands.

The following method, based on the generalized identification and counting technique of Section 2.2.3.2 is proposed for identifying Halstead operators and operands:

1. Comments are ignored.

2. Pragmas and declarations are included.

3. The following are counted as operators:

   All delimiters, including compound delimiters (e.g., ">=", ".."). For paired grouping symbols, such as "(" and ")", each pair is one operator.

   All reserved words.

4. The following are counted as operands:

   All literals, including numeric literals, character literals (inside the "'"s), and string literals (inside the """s).

   All other identifiers which are not reserved words in the context in which they are used. These include type names, formal parameter names, package names, function names, procedure names, task names, entry names, and exception names.

Note that since Ada allows the same name to refer to different entities, through scope changes and overloading, care must be taken to treat distinct entities as distinct operators and operands.

2.5.4.2 Possible Adjustments to the Counting Method

The counting method proposed above has been used on the simple examples in the following section, but it has not been verified by application to a statistically significant number of design problems and correlation with desired end products, such as estimates for development cost. When such data is collected, the counting method might need to be adjusted to perform better as a predictor of cost or yardstick of quality. This section lists some areas where the counting method might be adjusted.

The method used here is based on an expression of the design in pure Ada. If a superset is used, the new design language constructs will have to be included in the counting method. In particular, if Byron were used, the Ada comments which are Byron statements would need to be included in the metric determination, since the Byron statements are Ada comments. The Byron keywords and other constructs would need to be integrated with the counting for the Ada constructs.

The philosophy for dividing operators and operands might need to be reconsidered, especially in the areas of types, declaration versus use of items, task objects, generics, formal parameters, and renaming. [Els 78] discusses some of these issues in the context of PL/I.

Use of the use clause lets the designer use objects and operations from other packages without having to specify the full package and entity name in each reference. A design written this way will have lower total operator and, especially, operand counts. Fully qualified names might be a better reflection of the amount of effort expended by the designer. A metric evaluation tool using an APSE database would be able to resolve references to external packages, and compensate for the use of the use clause in the design.

2.5.4.3 Automation Potential

Automatic collection of the Halstead metric data from an Ada representation of a design is clearly feasible. It does, however, require a tool as complex as the syntactic and semantic analysis portions of a compiler, since the design is subject to the same potential ambiguities that a program is. Other problems may arise if the design language is a superset of Ada. For example, a superset might allow inclusion of free-form text as a significant part of the design. Such a tool would best be incorporated as part of an APSE, since the central database provides a natural means for tracking progress and monitoring quality as the development takes place.

2.5.4.4 Example

The first design problem in [Boo 83], counting the leaves on a binary tree, is used here as a sample for demonstrating the application of the Halstead metrics. The English language solution statement (the informal architectural design specification), the Ada architectural design specification, and the Ada detailed design specification are given in the figures which follow the discussion.

According to Halstead [Hal 77] Chapter 13, the software science parameters may be measured from the English solution statement by counting as operators the "function words" given in Table 13.1 ([Hal 77] p. 100), punctuation, capitalization, paragraphing, and numbers with one significant digit; and as operands all other words. The results of applying this counting method to the English statement of the informal problem solution for the counting leaves problem are given in Table 17 and Table 18.

Halstead also discusses adjusting for the redundancy usual in natural language, which comes about through using synonyms to make the prose style more interesting. He proposes that all distinct character patterns should be counted as distinct operators or operands and then the numbers for distinct operators and operands should be reduced by a factor to allow for synonyms and variant forms, such as plurals. He proposes 0.4 as a reasonable factor, and uses the difference between the estimated and actual lengths to substantiate this choice. Table 19 shows the values for the metrics with this adjustment made to the number of distinct operators and operands in Table 18. In this case, picking the factor to make the estimated length agree with the actual length of 132 gives a value of 0.554 for the factor. The metric values obtained by using this factor are also shown in Table 19.

Table 20 lists the operators and operands for the architectural level of the Ada design for the same problem, obtained by using the counting method defined in the previous section. Table 21 gives the Halstead metric values for the architectural design. Table 22 and Table 23 contain the operators, operands, and metric values for the detailed Ada design. In each of these figures, the information is shown for each major component of the system, as well as the system as a whole. That is, the COUNTER_PACKAGE, PILE_PACKAGE, TREE_PACKAGE, and the main procedure numbers are shown for the architectural design in the columns labelled CP, PP, TP, and main, respectively. The detailed design figures include, in addition, a column labelled FP, for the FIFO_PACKAGE.

2.5.4.5 Analysis of the Counting Leaves Example

This example shows that the Halstead metrics may be measured from an Ada design representation, but the utility of the data as a predicting and controlling tool remains to be verified on more, larger samples. As may be seen from the figures, the metric values increase with the addition of detail to the design, as expected. The correspondence between the program length estimator and the program length metric also improves with additional detail. It is likely that the metrics can be used to compare alternate designs, as long as they are compared at the same level of detail.

The Halstead metrics will favor the design which is expressed using fewer objects and operations, since they effectively correspond to the operands and operators counted in determining the Halstead measures. This encourages the designer to think about, and express, the solution in terms which are high level, as close as possible to the level of the problem statement, rather than in terms of the primitives of the system upon which it will be implemented. There is some evidence that the object-oriented design method encourages this kind of design, in that the estimated language levels for the architectural design are larger than those for the detailed design, and the estimated language level for the main subprogram is larger than those for the supporting packages.

Possibly, sets of metric values for components at different levels of detail could be used to develop a characteristic profile for how the values change as design progresses. The profile might then be used as a guideline for estimating development cost parameters for new modules, or for setting bounds on a reasonable growth rate for the metrics during the design process. Any module which exhibited an out-of-bounds growth rate would be suspect. The profile might depend on following a particular design method, such as the object-oriented method, in which the changes from level to level are well defined and constrained.

Keep a pile of the parts of the tree that have not yet been counted. Initially, get a tree and put it on the empty pile; the count of the leaves is initially set to zero. As long as the pile is not empty, repeatedly take a tree off the pile and examine it. If the tree consis of a single leaf, then increment the leaf counter and throw away that tree. If the tree is not a single leaf but instead consists of two subtrees split the tree into its left and right subtrees and put them back on the pile. Once the pile is empty, display the count of the leaves.

Figure 13. English Language Problem Statement for Counting Leaves

```
package COUNTER_PACKAGE is
  type COUNTER_TYPE is limited private;
  procedure DISPLAY (COUNTER : in COUNTER_TYPE);
  procedure INCREMENT (COUNTER : in out COUNTER_TYPE);
  procedure ZERO (COUNTER : out COUNTER_TYPE);
private
  ...
end COUNTER_PACKAGE;

with TREE_PACKAGE;
package PILE_PACKAGE is
  type PILE_TYPE is limited private;
  function IS_NOT_EMPTY (PILE : in PILE_TYPE)
     return BOOLEAN;
  procedure PUT (TREE : in out TREE_PACKAGE.TREE_TYPE;
                           ON   : in out PILE_TYPE);
  procedure PUT_INITIAL (TREE : in out TREE_PACKAGE.TREE_TYPE;
                           ON   : in out PILE_TYPE);
  procedure TAKE (TREE : out TREE_PACKAGE.TREE_TYPE;
                           OFF  : in out PILE_TYPE);
private
  ...
end PILE_PACKAGE;

package TREE_PACKAGE is
  type TREE_TYPE is private;
  procedure GET_INITIAL (TREE : out TREE_TYPE);
  function IS_SINGLE_LEAF (TREE : in TREE_TYPE)
     return BOOLEAN;
  procedure SPLIT (TREE   : in out TREE_TYPE;
                            LEFT_INTO :    out TREE_TYPE;
                            RIGHT_INTO:    out TREE_TYPE);
  procedure THROW_AWAY (TREE : in out TREE_TYPE);
private
  ...
end TREE_PACKAGE;
```

Figure 14. Ada Architectural Design Specification for Counting Leaves.

```
with COUNTER_PACKAGE, PILE_PACKAGE, TREE_PACKAGE;
use COUNTER_PACKAGE, PILE_PACKAGE, TREE_PACKAGE;
procedure COUNT_LEAVES_ON_BINARY_TREE is
  LEAF_COUNT    : COUNTER_TYPE;
  LEFT_SUBTREE  : TREE_TYPE;
  PILE          : PILE_TYPE;
  RIGHT_SUBTREE : TREE_TYPE;
  TREE          : TREE_TYPE;
begin
  GET_INITIAL(TREE);
  PUT_INITIAL(TREE, ON => PILE);
  ZERO(LEAF_COUNT);
  while IS_NOT_EMPTY(PILE);
    loop
      TAKE(TREE, OFF => PILE);
      if IS_SINGLE_LEAF(TREE) then
        INCREMENT(LEAF_COUNT);
        THROW_AWAY(TREE);
      else
        SPLIT(TREE,
              LEFT_INTO => LEFT_SUBTREE,
              RIGHT_INTO => RIGHT_SUBTREE);
        PUT(LEFT_SUBTREE, ON => PILE);
        PUT(RIGHT_SUBTREE, ON => PILE);
      end if;
    end loop;
  DISPLAY (LEAF_COUNT);
end COUNT_LEAVES_ON_BINARY_TREE;
```

Figure 15. Ada Solution Statement for Counting Leaves.

```
package COUNTER_PACKAGE is
  type COUNTER_TYPE is limited private;
  procedure DISPLAY (COUNTER : in COUNTER_TYPE);
  procedure INCREMENT (COUNTER : in out COUNTER_TYPE);
  procedure ZERO (COUNTER : out COUNTER_TYPE);
private
  type COUNTER_TYPE is NATURAL;
end COUNTER_PACKAGE;

with TEXT_IO;
package body COUNTER_PACKAGE is

  procedure DISPLAY (COUNTER : in COUNTER_TYPE) is
     package COUNTER_IO is new TEXT_IO.INTEGER_IO(COUNTER_TYPE);
  begin
     COUNTER_IO.PUT(COUNTER);
  end DISPLAY;

  procedure INCREMENT (COUNTER : in out COUNTER_TYPE) is
  begin
     COUNTER := COUNTER + 1;
  end INCREMENT;

  procedure ZERO (COUNTER : out COUNTER_TYPE) is
  begin
     COUNTER := 0;
  end ZERO;
end COUNTER_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - COUNTER_PACKAGE. (Part 1 of 8)

```
with FIFO_PACKAGE;
with TREE_PACKAGE;
package PILE_PACKAGE is
  type PILE_TYPE is limited private;
  function IS_NOT_EMPTY (PILE : in PILE_TYPE)
     return BOOLEAN;
  procedure PUT (TREE : in out TREE_PACKAGE.TREE_TYPE;
                        ON   : in out PILE_TYPE);
  procedure PUT_INITIAL (TREE : in out TREE_PACKAGE.TREE_TYPE;
                        ON   : in out PILE_TYPE);
  procedure TAKE (TREE : out TREE_PACKAGE.TREE_TYPE;
                        OFF  : in out PILE_TYPE);
private
  package TREE_QUEUE is new FIFO_PACKAGE(TREE_TYPE);
  type PILE_TYPE is TREE_QUEUE.QUEUE_TYPE;
end PILE_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - PILE_PACKAGE. (Part 2 of 8)

```
with TREE_QUEUE;
package body PILE_PACKAGE is

  function IS_NOT_EMPTY (PILE : in PILE_TYPE)
     return BOOLEAN is
  begin
     return not TREE_QUEUE.IS_EMPTY(PILE);
  end IS_NOT_EMPTY;

  procedure PUT (TREE : in out TREE_PACKAGE.TREE_TYPE;
                        ON   : in out PILE_TYPE) is
  begin
     TREE_QUEUE.APPEND (ELEMENT => TREE, TO => ON);
  end PUT;

  procedure PUT_INITIAL (TREE : in out TREE_PACKAGE.TREE_TYPE;
                        ON   : in out PILE_TYPE)
     renames PUT;

  procedure TAKE (TREE : out TREE_PACKAGE.TREE_TYPE;
                        OFF  : in out PILE_TYPE) is
  begin
     TREE_QUEUE.TAKE (ELEMENT => TREE, OFF => OFF);
  end TAKE;

end PILE_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - PILE_PACKAGE. (Part 3 of 8)

```
package TREE_PACKAGE is
  type TREE_TYPE is private;
  procedure GET_INITIAL (TREE : out TREE_TYPE);
  function IS_SINGLE_LEAF (TREE : in TREE_TYPE)
     return BOOLEAN;
  procedure SPLIT (TREE    : in out TREE_TYPE;
                            LEFT_INTO :     out TREE_TYPE;
                            RIGHT_INTO:     out TREE_TYPE);
  procedure THROW_AWAY (TREE : in out TREE_TYPE);
private
  type NODE;
  type TREE_TYPE is access NODE;
  type NODE_VALUE_TYPE is STRING(1..10);
  type NODE is
     record
        LEFT  : TREE_TYPE;
        VALUE : NODE_VALUE_TYPE;
        RIGHT : TREE_TYPE;
     end record;
end TREE_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - TREE_PACKAGE. (Part 4 of 8)

```
with DIRECT_IO;
package body TREE_PACKAGE is

  procedure GET_INITIAL (TREE : out TREE_TYPE) is
     -- Assume that the tree information is in a direct
     -- access file.  Each record consists of the information
     -- for one node of the tree, consisting of the value for
     -- the node, the file index of the top node of the left
     -- subtree, and the file index of the top node of the
     -- right subtree, in that order.
     -- The file index for the subtree will be zero
     -- if the node is a leaf.
     -- The topnode for the entire tree is in record 1.

        type TREE_RECORD_TYPE;
        package TREE_IO is new DIRECT_IO(TREE_RECORD_TYPE);
        type TREE_RECORD_TYPE is
           record
              VALUE      : NODE_VALUE_TYPE;
              LEFT_INDEX : TREE_IO.POSITIVE_COUNT;
              RIGHT_INDEX: TREE_IO.POSITIVE_COUNT;
           end record;

        TREE_RECORD: TREE_RECORD_TYPE;
        DATA_FILE  : TREE_IO.FILE_TYPE;

     with TREE_IO;
     procedure GET_SUBTREE (TREE : out TREE_TYPE;
                            RECORD_INDEX: in TREE_IO.POSITIVE_COUNT;
                            FILE        : in TREE_IO.FILE_TYPE;
                            TREE_RECORD : in out TREE_RECORD_TYPE
                           ) is
```

Figure 16. Ada Detailed Design for Counting Leaves - TREE_PACKAGE. (Part 5 of 8)

```
    begin
       if RECORD_INDEX = TREE_IO.POSITIVE_COUNT(0)
       then
          null;
       else
          TREE_IO.READ(FILE, ITEM => TREE_RECORD,
             FROM => RECORD_INDEX);
          TREE := new NODE'(LEFT => null,
                            VALUE => TREE_RECORD.VALUE,
                            RIGHT => null);
          GET_SUBTREE(TREE => TREE.LEFT,
                      RECORD_INDEX => TREE_RECORD.LEFT_INDEX,
                      FILE => FILE,
                      TREE_RECORD => TREE_RECORD);
          GET_SUBTREE(TREE => TREE.RIGHT,
                      RECORD_INDEX => TREE_RECORD.RIGHT_INDEX,
                      FILE => FILE,
                      TREE_RECORD => TREE_RECORD);
       end if;
    end GET_SUBTREE;

 begin
    TREE_IO.OPEN (DATA_FILE,
          MODE => IN_FILE,
          NAME => "Put in implementation detail here",
          FORM => "Put in implementation detail here");
    GET_SUBTREE(TREE => TREE,
                RECORD_INDEX => TREE_IO.POSITIVE_COUNT(1),
                FILE => DATA_FILE,
                TREE_RECORD => TREE_RECORD);
    TREE_IO.CLOSE (DATA_FILE);
 exception
    -- Fill in exception processing during implementation.
    when DATA_ERROR =>
       null;
```

Figure 16. Ada Detailed Design for Counting Leaves - TREE_PACKAGE. (Part 6 of 8)

```
      when DEVICE_ERROR =>
         null;
      when END_ERROR =>
         null;
      when NAME_ERROR =>
         null;
      when STATUS_ERROR =>
         null;
      when USE_ERROR =>
         null;
   end GET_INITIAL;

   function IS_SINGLE_LEAF (TREE : in TREE_TYPE)
      return BOOLEAN is
   begin
      return (TREE.LEFT = null) and (TREE.RIGHT = null);
   end IS_SINGLE_LEAF;

   procedure SPLIT (TREE   : in out TREE_TYPE;
                      LEFT_INTO :    out TREE_TYPE;
                      RIGHT_INTO:    out TREE_TYPE) is
   begin
      LEFT_INTO  := TREE.LEFT;
      RIGHT_INTO := TREE.RIGHT;
      THROW_AWAY(TREE);
   end SPLIT;

   procedure THROW_AWAY (TREE : in out TREE_TYPE) is
   begin
      -- Assume that deallocation and garbage collection are
      -- done by the system.
      TREE := null;
   end THROW_AWAY;
end TREE_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - TREE_PACKAGE. (Part 7 of 8)

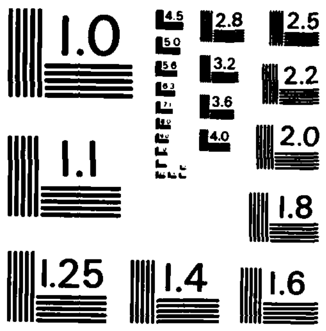
```
generic
  type QUEUE_ELEMENT_VALUE_TYPE is private;
package FIFO_PACKAGE is
  type QUEUE_TYPE is limited private;
  EMPTY_QUEUE: constant QUEUE_TYPE;
  function "=" (QUEUE1: in QUEUE_TYPE;
                       QUEUE2: in QUEUE_TYPE)
     return BOOLEAN;
  function IS_EMPTY (QUEUE : in QUEUE_TYPE)
     return BOOLEAN;
  procedure APPEND (ELEMENT: in QUEUE_ELEMENT_VALUE_TYPE;
                       TO     : in out QUEUE_TYPE);
  procedure TAKE (ELEMENT: out QUEUE_ELEMENT_VALUE_TYPE;
                       OFF    : in out QUEUE_TYPE);
private
  type QUEUE_ELEMENT_TYPE;
  type QUEUE_TYPE is access QUEUE_ELEMENT_TYPE;
  type QUEUE_ELEMENT_TYPE is
     record
        VALUE: QUEUE_ELEMENT_VALUE_TYPE;
        REST : QUEUE_TYPE;
     end record;
end FIFO_PACKAGE;
```

Figure 16. Ada Detailed Design for Counting Leaves - FIFO_PACKAGE. (Part 8 of 8)

AD-A145 869 AUTOMATING SOFTWARE DESIGN METRICS(U) CHARLES STARK DRAPER LAB INC CAMBRIDGE MA P A SZULEWSKI ET AL. FEB 84 CSDL-R-1662 RADC-TR-84-27 F30602-82-C-0130 2/2

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

Table 17. Operators and Operands in English Statement for Counting Leaves

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - ENGLISH

OPERATORS	COUNT	OPERANDS	COUNT
<paragraph>	1	off	1
<upper case>	6	put	2
.	6	set	1
,	5	back	1
;	1	leaf	3
a	5	left	1
as	2	long	1
if	2	pile	6
is	4	take	1
it	2	tree	7
of	6	count	2
on	2	empty	3
to	1	parts	1
and	5	split	1
but	1	throw	1
get	1	leaves	2
its	1	single	2
not	3	counted	1
the	15	counter	1
two	1	display	1
yet	1	examine	1
away	1	consists	2
been	1	subtrees	2
have	1	increment	1
into	1	initially	2
keep	1	repeatedly	1
once	1		
that	2		
them	1		
then	1		
zero	1		
right	1		
instead	1		
33	84	26	48

Table 18. Halstead Metric Values for English Statement for Counting Leaves

HALSTEAD METRIC	[BOO 83] DESIGN PROBLEM ONE - ENGLISH
DISTINCT OPERATORS	33
DISTINCT OPERANDS	26
TOTAL OPERATORS	84
TOTAL OPERANDS	48
VOCABULARY	59
DESIGN LENGTH	132
ESTIMATED LENGTH	289
PERCENT OFF	-119
DESIGN VOLUME	777
POTENTIAL VOLUME	26
ESTIMATED DESIGN LEVEL	0.03
INTELLIGENCE CONTENT	25
ESTIMATED LANGUAGE LEVEL	0.8
ESTIMATED EFFORT	23,654

Table 19. Halstead Metric Values for English Statement Adjusted for Redundancy

HALSTEAD METRIC	[BOO 83] DESIGN PROBLEM ONE - ENGLISH	
	Redundancy factor = 0.4	0.554
DISTINCT OPERATORS	13.2	18.3
DISTINCT OPERANDS	10.4	14.4
TOTAL OPERATORS	84	84
TOTAL OPERANDS	48	48
VOCABULARY	23.6	32.7
DESIGN LENGTH	132	132
ESTIMATED LENGTH	84	132
PERCENT OFF	36	-0.1
DESIGN VOLUME	602	664
POTENTIAL VOLUME	20	22
ESTIMATED DESIGN LEVEL	0.03	0.03
INTELLIGENCE CONTENT	20	22
ESTIMATED LANGUAGE LEVEL	0.6	0.7
ESTIMATED EFFORT	18,338	20,256

Table 20. Operators and Operands in Architectural Design for Counting Leaves. (Part 1 of 3)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - ARCH DES					
OPERATORS	CP	PP	TP	main	system
.	0	3	0	0	3
( )	3	4	4	12	23
;	5	10	8	21	44
,	0	0	0	10	10
:	3	7	6	5	21
=>	0	0	0	6	6
begin	0	0	0	1	1
else	0	0	0	1	1
end	1	1	1	3	6
function	0	1	1	0	2
if	0	0	0	2	2
in	2	6	3	0	11
is	2	2	2	1	7
limited	1	1	0	0	2
loop	0	0	0	2	2
out	2	6	5	0	13
package	1	1	1	0	3
private	2	2	2	0	6
procedure	3	3	3	1	10
return	0	1	1	0	2
then	0	0	0	1	1
type	1	1	1	0	3
use	0	0	0	1	1
while	0	0	0	1	1
with	0	1	0	1	2
DISTINCT OPERATORS	12	16	13	16	25
TOTAL OPERATORS	26	50	38	69	183

Table 20. Operators and Operands in Architectural Design for Counting Leaves. (Part 2 of 3)

HALSTEAD METRIC	[BOO 83] DESIGN PROBLEM ONE - ARCH DES			
OPERANDS	CP	PP	TP	main	system

OPERANDS	CP	PP	TP	main	system
BOOLEAN	0	1	1	0	2
COUNT_LEAVES_ON_BINARY_TREE	0	0	0	2	2
COUNTER(DISPLAY)	1	0	0	0	1
COUNTER(INCREMENT)	1	0	0	0	1
COUNTER(ZERO)	1	0	0	0	1
COUNTER_PACKAGE	2	0	0	2	4
COUNTER_TYPE	4	0	0	1	5
DISPLAY	1	0	0	1	2
GET_INITIAL	0	0	1	1	2
INCREMENT	1	0	0	1	2
IS_NOT_EMPTY	0	1	0	1	2
IS_SINGLE_LEAF	0	0	1	1	2
LEAF_COUNT	0	0	0	4	4
LEFT_INTO	0	0	1	1	2
LEFT_SUBTREE	0	0	0	3	3
OFF(PP.TAKE)	0	1	0	1	2
ON(PUT)	0	1	0	2	3
ON(PUT_INITIAL)	0	1	0	1	2
PILE(main)	0	0	0	6	6
PILE(IS_NOT_EMPTY)	0	1	0	0	1
PILE_PACKAGE	0	2	0	2	4
PILE_TYPE	0	5	0	1	6
PUT(PILE_PKG)	0	1	0	2	3
PUT_INITIAL	0	1	0	1	2
RIGHT_INTO	0	0	1	1	2

Table 20. Operators and Operands in Architectural Design for Counting Leaves. (Part 3 of 3)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - ARCH DES

OPERANDS	CP	PP	TP	main	system
RIGHT_SUBTREE	0	0	0	3	3
SPLIT	0	0	1	1	2
TAKE(PP)	0	1	0	1	2
THROW_AWAY	0	0	1	1	2
TREE(PUT)	0	1	0	0	1
TREE(PUT_INITIAL)	0	1	0	0	1
TREE(TAKE)	0	1	0	0	1
TREE(GET_INITIAL)	0	0	1	0	1
TREE(IS_SINGLE_LEAF)	0	0	1	0	1
TREE(SPLIT)	0	0	1	0	1
TREE(THROW_AWAY)	0	0	1	0	1
TREE(main)	0	0	0	7	7
TREE_PACKAGE	0	4	2	2	8
TREE_TYPE	0	3	7	3	13
ZERO	1	0	0	1	2
DISTINCT OPERANDS	8	16	13	28	40
TOTAL OPERANDS	12	26	20	54	112

Table 21. Halstead Metric Values for Architectural Design for Counting Leaves

HALSTEAD METRIC	[BOO 83] DESIGN PROBLEM ONE - ARCH DES				
	CP	PP	TP	main	system
DISTINCT OPERATORS	12	16	13	16	25
DISTINCT OPERANDS	8	16	13	28	40
TOTAL OPERATORS	26	50	38	69	183
TOTAL OPERANDS	12	26	20	54	112
VOCABULARY	20	32	26	44	65
DESIGN LENGTH	38	76	58	123	295
ESTIMATED LENGTH	67	128	96	199	329
PERCENT OFF	-76	-68	-66	-61	-12
DESIGN VOLUME	164	380	273	672	1777
ESTIMATED DESIGN LEVEL	0.1	0.1	0.1	0.1	0.03
INTELLIGENCE CONTENT	18	29	27	44	51
ESTIMATED LANG LEVEL	2.0	2.2	2.7	2.8	1.5
ESTIMATED EFFORT	1478	4940	2726	10360	62181

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 1 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES

OPERATORS	CP	PP	TP	main	FP	system
'	0	0	1	0	0	1
"	0	0	2	0	1	3
.	2	10	19	0	0	31
( )	8	12	23	12	4	59
;	15	25	55	21	16	132
,	0	2	16	10	0	28
:	6	14	24	5	10	59
+	1	0	0	0	0	1
=(relational)	0	0	3	0	0	3
=(fcn name)	0	0	0	0	1	1
..	0	0	1	0	0	1
=>	0	4	26	6	0	36
:=	2	0	4	0	0	6
access	0	0	1	0	1	2
and	0	0	1	0	0	1
begin	3	3	5	1	0	12
body	1	1	1	0	0	3
constant	0	0	0	0	1	1
else	0	0	1	1	0	2
end	5	5	10	3	2	25
exception	0	0	1	0	0	1
function	0	2	2	0	2	6
generic	0	0	0	0	1	1
if	0	0	2	2	0	4
in	4	12	9	0	6	31
is	8	8	12	1	5	34
limited	1	1	0	0	1	3
loop	0	0	0	2	0	2
new	1	1	2	0	0	4

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 2 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES						
OPERATORS	CP	PP	TP	main	FP	system
not	0	1	0	0	0	1
null(statement)	0	0	7	0	0	7
null(access value)	0	0	5	0	0	5
out	4	12	12	0	3	31
package	3	3	3	0	1	10
private	2	2	2	0	3	9
procedure	6	6	7	1	2	22
record	0	0	4	0	2	6
renames	0	1	0	0	0	1
return	0	3	3	0	2	8
then	0	0	1	1	0	2
type	2	2	6	0	5	15
use	0	0	0	1	0	1
when	0	0	6	0	0	6
while	0	0	0	1	0	1
with	1	3	2	1	0	7
DISTINCT OPERATORS	19	23	35	16	20	45
TOTAL OPERATORS	75	133	279	69	69	625

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 3 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES

OPERANDS	CP	PP	TP	main	FP	system
APPEND	0	1	0	0	1	2
BOOLEAN	0	2	2	0	2	6
CLOSE	0	0	1	0	0	1
COUNT_LEAVES_ON_ BINARY_TREE	0	0	0	2	0	2
COUNTER(DISPLAY)	3	0	0	0	0	3
COUNTER(INCREMENT)	4	0	0	0	0	4
COUNTER(ZERO)	3	0	0	0	0	3
COUNTER_IO	2	0	0	0	0	2
COUNTER_PACKAGE	4	0	0	2	0	6
COUNTER_TYPE	9	0	0	1	0	10
DATA_ERROR	0	0	1	0	0	1
DATA_FILE	0	0	4	0	0	4
DEVICE_ERROR	0	0	1	0	0	1
DIRECT_IO	0	0	2	0	0	2
DISPLAY	3	0	0	1	0	4
ELEMENT(TQ.APPEND)	0	1	0	0	1	2
ELEMENT(TQ.TAKE)	0	1	0	0	1	2
EMPTY_QUEUE	0	0	0	0	1	1
END_ERROR	0	0	1	0	0	1
FIFO_PACKAGE	0	2	0	0	2	4
FILE	0	0	7	0	0	7
FILE_TYPE	0	2	0	0	0	2
FORM	0	0	1	0	0	1
FROM	0	0	1	0	0	1
GET_INITIAL	0	0	3	1	0	4
GET_SUBTREE	0	0	5	0	0	5
IN_FILE	0	0	1	0	0	1
INCREMENT	3	0	0	1	0	4
INTEGER_IO	1	0	0	0	0	1

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 4 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES

OPERANDS	CP	PP	TP	main	FP	system
IS_EMPTY	0	1	0	0	1	2
IS_NOT_EMPTY	0	3	0	1	0	4
IS_SINGLE_LEAF	0	0	3	1	0	4
ITEM	0	0	1	0	0	1
LEAF_COUNT	0	0	0	4	0	4
LEFT	0	0	5	0	0	5
LEFT_INDEX	0	0	2	0	0	2
LEFT_INTO	0	0	3	1	0	4
LEFT_SUBTREE	0	0	0	3	0	3
MODE	0	0	1	0	0	1
NAME	0	0	1	0	0	1
NAME_ERROR	0	0	1	0	0	1
NATURAL	1	0	0	0	0	1
NODE	0	0	4	0	0	4
NODE_VALUE_TYPE	0	0	3	0	0	3
OFF(PP.TAKE)	0	3	0	1	0	4
OFF(TQ.TAKE)	0	1	0	0	0	1
OFF(FIFO.TAKE)	0	0	0	0	1	1
ON(PUT)	0	3	0	2	0	5
ON(PUT_INITIAL)	0	2	0	1	0	3
OPEN	0	0	1	0	0	1
PILE(CLOBT)	0	0	0	6	0	6
PILE(IS_NOT_EMPTY)	0	3	0	0	0	3
PILE_PACKAGE	0	4	0	2	0	6
PILE_TYPE	0	10	0	1	0	11
POSITIVE_COUNT	0	0	5	0	0	5
PUT(PILE_PKG)	0	4	0	2	0	6
PUT(COUNTER_IO)	1	0	0	0	0	1
PUT_INITIAL	0	2	0	1	0	3
QUEUE	0	0	0	0	1	1

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 5 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES

OPERANDS	CP	PP	TP	main	FP	system
QUEUE1	0	0	0	0	1	1
QUEUE2	0	0	0	0	1	1
QUEUE_ELEMENT_ VALUE_TYPE	0	0	0	0	4	4
QUEUE_ELEMENT_TYPE	0	0	0	0	3	3
QUEUE_TYPE	0	1	0	0	9	10
READ	0	0	1	0	0	1
RECORD_INDEX	0	0	6	0	0	6
REST	0	0	0	0	1	1
RIGHT	0	0	5	0	0	5
RIGHT_INDEX	0	0	2	0	0	2
RIGHT_INTO	0	0	3	1	0	4
RIGHT_SUBTREE	0	0	0	3	0	3
SPLIT	0	0	3	1	0	4
STATUS_ERROR	0	0	1	0	0	1
STRING	0	0	1	0	0	1
TAKE(PP)	0	3	0	1	0	4
TAKE(TQ)	0	1	0	0	0	1
TAKE(FIFO)	0	0	0	0	1	1
TEXT_IO	2	0	0	0	0	2
THROW_AWAY	0	0	4	1	0	5
TO(APPEND)	0	1	0	0	1	2
TREE(PUT)	0	3	0	0	0	3
TREE(PUT_INITIAL)	0	2	0	0	0	2
TREE(TAKE)	0	3	0	0	0	3
TREE(GET_INITIAL)	0	0	3	0	0	3
TREE(GET_SUBTREE)	0	0	7	0	0	7
TREE(IS_SINGLE_LEAF)	0	0	4	0	0	4
TREE(SPLIT)	0	0	5	0	0	5
TREE(THROW_AWAY)	0	0	3	0	0	3

Table 22. Operators and Operands in Detailed Design for Counting Leaves. (Part 6 of 6)

HALSTEAD METRIC [BOO 83] DESIGN PROBLEM ONE - DET DES						
OPERANDS	CP	PP	TP	main	FP	system
TREE(mainproc)	0	0	0	7	0	7
TREE_IO	0	0	12	0	0	12
TREE_PACKAGE	0	7	4	2	0	13
TREE_RECORD_TYPE	0	0	5	0	0	5
TREE_RECORD (GET_INITIAL)	0	0	2	0	0	2
TREE_RECORD (GET_SUBTREE)	0	0	10	0	0	10
TREE_QUEUE	0	6	0	0	0	6
TREE_TYPE	0	7	17	3	0	27
USE_ERROR	0	0	1	0	0	1
VALUE(FIFO)	0	0	0	0	1	1
VALUE(NODE)	0	0	2	0	0	2
VALUE (TREE_RECORD_TYPE)	0	0	2	0	0	2
ZERO	3	0	0	1	0	4
0	1	0	1	0	0	2
1	1	0	2	0	0	3
10	0	0	1	0	0	1
Put in implementation detail here	0	0	2	0	0	2
DISTINCT OPERANDS	15	27	52	28	18	105
TOTAL OPERANDS	41	79	169	54	33	376

Table 23. Halstead Metric Values for Detailed Design for Counting Leaves

HALSTEAD METRIC	[BOO 83] DESIGN PROBLEM ONE - DET DES					
	CP	PP	TP	main	FP	system
DISTINCT OPERATORS	19	23	35	16	20	45
DISTINCT OPERANDS	15	27	52	28	18	105
TOTAL OPERATORS	75	133	279	69	69	625
TOTAL OPERANDS	41	79	169	54	33	376
VOCABULARY	34	50	87	44	38	150
DESIGN LENGTH	116	212	448	123	102	1001
ESTIMATED LENGTH	139	232	476	199	161	952
PERCENT OFF	-20	-10	-6	-61	-58	5
DESIGN VOLUME	590	1196	2886	672	535	7236
ESTIMATED DESIGN LEVEL	0.04	0.03	0.02	0.06	0.05	0.01
INTELLIGENCE CONTENT	23	36	51	44	29	90
ESTIMATED LANG LEVEL	0.9	1.1	0.9	2.8	1.6	1.1
ESTIMATED EFFORT	62181	40260	164166	10360	9814	583018

102 BlANK

3.0 USING DESIGN METRICS, A SUPPORTING METHODOLOGY

This section outlines a methodology, consistent with the RADC software quality framework, for using design-aid tools and design metrics. It envisions a situation in which such tools and metrics are part of an integrated software engineering environment which is used to support all phases of software development. Using this approach, it is possible to

1. Evaluate competing software designs,

2. Estimate software project planning parameters,

3. Monitor software product quality.

Previous sections of this report have described and illustrated various software design media, and have shown how metrics can be defined for measuring certain software quality criteria. In particular, the CSDL tool DARTS was introduced and used to illustrate the automatic generation of McCabe and Halstead metrics based on the information in a DARTS design database. An example was also given of the use of an Ada PDL as a design medium, and Halstead metrics were extracted in a similar fashion. Both the Halstead and McCabe metrics provide a means for assessing the complexity (or inversely, simplicity) of competing designs. The Halstead metric is also capable of indicating conciseness of a design, as discussed previously.

This section begins with the definition of a method for projecting project costs and schedules based on metric data taken during the early phases of a project. It then describes in more detail the anticipated use of design metrics by both software development and program office personnel.

3.1 PROJECTING PROJECT COSTS AND SCHEDULES

To estimate costs and schedules in the early phases of a software development project, one must adopt a cost estimation method, gather any needed data, estimate required parameters, and carry out the required computations. Boehm reviews a number of cost estimation methods [Boe 81], and recommends using a combination of techniques including:

- Top-down estimates based on expert opinion and previous experience, and
- Bottom-up estimates (module by module) using an algorithmic model.

Although there is no substitute for expert opinion and previous experience, algorithmic models have been found increasingly useful as a base for estimating project cost and schedule. Boehm's COCOMO model [Boe 81] is an example of an estimation method that has been widely used and adopted by many organizations. It has the advantage of being well documented, and has been validated using data collected from 63 completed projects. As an example, the COCOMO intermediate model estimated software cost within 20 percent of project actual cost 68 percent of the time.

3.1.1 An Algorithmic Estimation Method

Halstead presents an algorithmic estimation method which has certain desirable features, although it has been criticized for some of its assumptions. It is based on two aspects of software science theory:

- The effort measure E, and its relation to programming effort, and
- The potential volume V*, and the ability to infer software science metrics from V* and the implementation language level λ .

Halstead suggests that once E is known, an estimate for the programming effort can be obtained from:

$$T = E/S$$

where T is programming time and S is the Stroud number. Halstead used the Stroud number as a measure of the number of elementary mental discriminations a programmer would make per unit time. He cited the range for S as from 5 to 20 discriminations per second, and most commonly used 18 discriminations per second (in which case T is measured in seconds). This approach has been criticized recently as an incorrect application of the results of cognitive psychology studies [Cou 83], although a certain amount of empirical evidence has been amassed in its favor.

To obtain an estimate of E for an implementation, Halstead suggests using the potential volume V* and the implementation language level λ . The potential volume V* is a measure of the volume of an algorithm in its minimal form, namely, a "built-in" function which computes the algorithm from a list of its input and output parameters. If the number of such parameters (operators) is designated η_2^* , then the minimal vocabulary η^* is computed as

$$\eta^* = 2 + \eta_2^*$$

where the operand count is taken as 2 to account for the name of the procedure and the assignment operator (or grouping symbol). The potential volume is computed from

$$V^* = \eta^* \log_2 \eta^*$$

The language level λ was proposed as a parameter that would characterize a programming language in terms of expressive power. Halstead defined it in terms of the program level L and the volume V as

$$\lambda = L^2 V = LV^*$$

By analyzing programs written in a number of different languages, he was able to measure λ as 1.53 for PL/1, 1.14 for Fortran, and 0.88 for CDC assembly language, although these values had large variances. Subsequent research has failed to corroborate these results for other sets of data, and the claim for constancy of the language level must now be regarded as questionable [She 83].

Nevertheless, Halstead gives the following formula for estimating E [Hal 77]

$$\hat{E} = (V^*)^3/\lambda^2$$

In other words, by estimating λ for the implementation language and by knowing V* from a count of the inputs and outputs of the algorithm, one can estimate E for the implementation, and then calculate T.

In the following, it is proposed to use the potential volume and language level in order to estimate the program length N instead of E, and thus avoid use of the controversial Stroud number. Although a value must be selected for the language level, it does not otherwise enter into the method and in particular does not need to be measured. The utility of the procedure must be empirically determined.

The program length N can be directly related to length in thousands of delivered source instructions (KDSI), and hence estimates of cost and schedule can be derived using Boehm's COCOMO model or other models. For example, in the basic COCOMO model

$$MM = 2.4\ (KDSI)^{1.05}$$

$$TDEV = 2.5\ (MM)^{0.38}$$

where MM is effort in man-months and TDEV is development time in months, and the so-called organic development mode is assumed (see [Boe 81]). Note that in talking about delivered source instructions the following comments apply.

1. Delivered means any software developed with the same rigor as as a deliverable product, e.g., software developed with reviews, test plans, documentation, etc.

2. Source instructions exclude comments but include job control language, format statements, and data declarations.

Note also that the above formulas exclude the effort required for the plans and requirements phase, and that good management practices are assumed.

Halstead's length N may be related to KDSI as follows:

$$N = a \times b \times KDSI \times 10^3$$

where a is the ratio of executable to total source statements (since only executable source statements are counted in Halstead's method), and b is the number of operators and operands in one source statement. Halstead suggested using .5 for a, and for b, 7.5 for a high level language and 2.7 for assembly language [Hal 77, Hal 78a]. Using the value for a high level language, one finds

$$KDSI = (2.667 \times 10^{-4})\ N$$

To find an estimate of the length N, one first finds the volume estimate using Halstead's relation [Hal 77]

$$V = (V^*)^2/2$$

By definition, the volume is related to the length

$$V = N \log_2 (\eta/2)$$

where η is the vocabulary. Employing the approximation, $\eta_1 = \eta_2 = \eta/2$, one has

$$N \simeq \eta \log_2 (\eta)$$

Thus for a given V* and λ , one must find η such that

$$f(\eta) = \eta \log_2(\eta/2) \log_2\eta - (V^*)^2/\lambda = 0$$

Once η is found, then N is found from the preceeding formula.

Gaffney presents an alternative method for estimating N that has the advantage of not involving the language level [Gaf 81], but it appears to be of use only for relatively large modules.

One can apply this method to the design taken as a whole, but it is preferable to use data from the architectural design to estimate the length of each of the modules separately, and then add to get the total design length. Thus, the technique provides initial estimates for the length (and hence the project cost and schedule) which are refined and improved as the design proceeds. The following example will make this clearer.

3.1.2 An Example

To illustrate this method, the experiment controller example of section 2.4 will be used. The input and output data items for each node in the design are stored in the DARTS database, so that it is a simple matter to determine η^* and hence V^*. For this example, λ was taken to be 1.3, corresponding to a high level implementation language. The data for design one are presented in Table 24, and for design two in Table 25.

Referring to Table 24, it is seen that when the counting method was applied to the top node in the design (component 1.1), N was found to be 300.2 which corresponds to about 80 lines of source code. When the method is applied to the next level of the design, and the lines of code estimates are totalled, an estimate of 110 lines of code is obtained. Similarly, when the method is applied to three levels, the estimate is reduced to 95 lines of code.

Referring to Table 25, the top level estimate is the same, namely, 80 lines of code. However, the second and third level estimates are 165 and 167 lines respectively. These estimates reinforce the conclusions reached earlier, namely, that design two is more complex than design one. Furthermore, they provide insight into the relative cost of implementing the two approaches. The result is especially striking, since to three levels, design two has only six components, while design one has eleven.

The estimates also enable the designer to identify the sources of complexity in the design. When estimates jump from one level to the next (as with design two), it signals an unexpectedly large increase in complexity. The designer can then identify the modules which most contribute to the complexity, and determine if improvements can be made.

A final point worth mentioning is that this technique depends on being able to determine η^* and hence V^* for designs and design components. In particular, if data item names represent abstractions for complicated structures, the number of unique operands needs to be appropriately increased. This can

Table 24. Length Estimates for Design One

Component	η^*	V^*	η	N	KDSI	Σ KDSI
1.1	13	48.106	60.9	300.2	0.080	0.080
1.1.1	3	4.755	5.24	7.3	0.002	0.110
1.1.2	5	11.610	11.6	29.3	0.008	
1.1.3	14	53.303	69.6	358.	0.096	
1.1.4	4	8.000	8.09	16.3	0.004	
1.1.1	3	4.755	5.24	7.3	0.002	0.095
1.1.2.1	4	8.000	8.09	16.3	0.004	
1.1.2.2	5	11.610	11.6	29.3	0.008	
1.1.3.1	6	15.510	15.7	46.6	0.012	
1.1.3.2	5	11.610	11.6	29.3	0.008	
1.1.3.4	4	8.000	8.09	16.3	0.004	
1.1.3.5	9	28.529	31.5	125.3	0.033	
1.1.3.6	5	11.610	11.6	29.3	0.008	
1.1.4.1	4	8.000	8.09	16.3	0.004	
1.1.4.2	5	11.610	11.6	29.3	0.008	
1.1.4.3	4	8.000	8.09	16.3	0.004	

probably be done without too much difficulty, but further experience with the technique is needed in order suggest practical methods for handling this situation.

3.2 USE OF DESIGN METRICS

Although design metrics can be used whenever design information is available, major improvements in being able to monitor and influence software quality will only occur when design tools and metrics are made part of an integrated software engineering environment. The various Ada Programming Support Environments (APSEs) now under development provide an ideal opportunity for application of these techniques. Although the metrics would provide information of primary use to developers, program office personnel would, using suitable contract clauses and data item descriptions, be able to request quality status information at periodic intervals based on information in the project data base. This would be greatly facilitated if standards existed for generating the required information.

Table 25. Length Estimates for Design Two

Component	η^*	V*	η	N	KDSI	Σ KDSI
2.1	13	48.106	60.9	300.2	0.080	0.080
2.1.1	6	15.510	15.7	46.6	0.012	0.165
2.1.2	17	69.487	99.3	559.	0.149	
2.1.3	4	8.000	8.09	16.3	0.004	
2.1.1	6	15.510	15.7	46.6	0.012	0.167
2.1.2.1	10	33.219	38.0	161.6	0.043	
2.1.2.2	14	53.303	69.6	358.	0.096	
2.1.3.1	4	8.000	8.09	16.3	0.004	
2.1.3.2	5	11.610	11.6	29.3	0.008	
2.1.3.3	4	8.000	8.09	16.3	0.004	

The uses of design metrics in the three earliest phases of the life cycle can be outlined as follows:

1. Software Requirements Specification

- Requirements are generated using a suitable requirements specification language or tool. If the requirements are expressed as a hierarchy of functions with inputs and outputs specified (as per MIL-STD-483/490, type B5), then Halstead metrics can be applied. Alternatively, it is possible to develop a Halstead technique using a prose requirements specification, adapting the form used in Section 2.5.4.4.

- These metrics would be used to generate estimates of KDSI for comparison with KDSI estimates generated by conventional means.

- Particularly complicated functions would be identified for risk assessment.

- Review of these estimates would be a topic included at the Software Requirements Review.

2. Architectural Design

- Structured analysis (or some other technique) is used to identify objects for a DARTS process-based model, or an Ada object-oriented design. The design is entered into a design database. At frequent and periodic intervals, Halstead, McCabe, and other design metrics are applied to the design data.
- The metrics are used to compare different design approaches in terms of complexity and impact on project cost and schedule.
- The metrics are used to identify problematic, overly complex, or inconcise areas of the design, in order to make improvements.
- The metrics are used to monitor the progress of the design as time proceeds.
- Review of these estimates would be a topic included at the Preliminary Design Review.

3. Detailed Design

- The detailed design would proceed as a refinement of the architectural design. The metrics would continue to be used in a similar fashion.
- Review of the metrics data would be a topic included at the Critical Design Review.

It is anticipated that the chief value of the design metrics would be to call attention to the impact of design decisions as they are made. The metrics would supplement but in no way replace the judgment of experienced software designers and managers. The ability to measure the impacts of design decisions and to institute corrective actions early in the development process would be a sign that a major improvement had indeed been made in the manner in which software products are designed and implemented.

4.0 CONCLUSIONS

This research effort has concluded the following.

1. Software quality can be assessed early in the life-cycle with metrics like those of McCabe and Halstead. The collection of metric data can be automated and captured from the databases of design tools like DARTS. However, the databases must contain the information necessary for metric measurement in a form recognizable by the design tool. For example, control flow or data definitions which occur in free-form text will not be included in metric calculation.

2. Out of the approximately 100 McCall metric elements, about 25% are good prospects for automatic measurement with a design tool like DARTS. However, manual assistance is required to measure most criteria in the framework since the criteria contain some subjective elements.

3. Metrics that measure data elements or some aspect of data-flow are generally more useful than those that deal with control flow, in the early design phases. Interfaces between software components are typically defined prior to the internal control structure of the components. This would include, for example, Halstead's metrics.

4. When Halstead metrics are used during design and the Potential Volume of each component is known, alternative counting techniques can be developed which avoid some of the controversial issues surrounding the Halstead counting technique.

5. Design metrics can be used when Ada or an Ada PDL is used during design. Ada supersets, like Byron, are more useful than Ada itself during architectural design due to their ability to represent abstract concepts and include prose commentary.

6. It is still early to judge the utility of design metrics. Small examples seem to work, but the size of the sample used for validation was too small. A more rigorous validation effort might be set up as follows: During a real software development effort, use metrics with an automated tool and sample software quality at frequent intervals during development to get a better idea of how useful early data is for estimating the quality of delivered software products.

7. A methodology has been presented for the use of quality metrics during design. In particular, it is possible to use counts of operators and operands to estimate the length of an implementation in terms of delivered source instructions. From this parameter, project costs and sche-

dules can be estimated. These estimates can be reevaluated as the design proceeds, in order to monitor software product quality.

5.0 DIRECTIONS FOR FURTHER RESEARCH

As this study has shown, design metrics have the potential for improving control over the software development process. This section lists areas in which future research would enhance the practicality of using design metrics.

The prime need is for verification of the metrics from metric data collected throughout the life-cycle of many real projects. This would provide a statistically significant database from which equations which relate metric values, planning parameters, and quality factors could be drawn. The examples provided in the preceeding text indicate that the metrics could prove useful, but they exhibit pathological behavior, such as extreme sensitivity to small changes, which would be eliminated with a larger sample. Incorporating metric tools into an APSE would encourage data collection.

Another area where work is needed is in relating the Software Quality Framework to other, more direct, measures of the quality factors. For example, reliability as measured in terms of mean time to next failure.

This research identified controversial issues surrounding the Halstead identification and counting technique. This evidence, as well as other research, establishes the need to develop a consistent Halstead counting method for use throughout design to deal with problems where the Potential Volume is not known.

This report has addressed metric measurement of design complexity. Metrics could be developed which evaluate the efficacy of the design procedures: how the development methods used contribute to the quality factors. Similarly, quality measurement could be expanded to cover the entire software, and system development processes. A key support item for these endeavors is an integrated database of real project data, from planning through maintenance, with data collection for quality evaluation as an integral part of the process.

Another area which might be covered in more depth is how tradeoffs among the quality factors may be measured and evaluated. This task was within the original scope of this project, but it was not initiated due to a decision to consider Ada design metrics in detail.

Work should be undertaken to establish the best use of requirements and design media for Ada. Ada has growing potential as a software design aid itself. This task should consider whether a restricted subset of Ada, Ada as defined, or Ada with additional features (e.g., Byron) is more useful and appropriate as a design or requirements aid.

114 BLANK

APPENDIX A. GLOSSARY OF RELATED ACRONYMS AND TERMS

APSE	Ada Programming Support Environment
AMT	Automatic Measurement Tool
CACM	Communications of the ACM
CDC	Control Data Corporation
CDR	Critical Design Review
COCOMO	COnstructive COst MOdel
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSDL	The Charles Stark Draper Laboratory, Inc.
DARTS	Design Aids for Real-Time Systems
DID	Data Item Description
DOD	Department of Defense
ECSL	Extended Control and Simulation Language
ESD	Electronic Systems Division
HOL	Higher Order Language
KDSI	Thousands of Delivered Source Instructions
PDL	Program Design Language
PDR	Preliminary Design Review
PERT	Program Evaluation and Review Technique
QA	Quality Assurance
RADC	Rome Air Development Center
SQA	Software Quality Assurance

SRR	Software Requirements Review
STARS	Software Technology for Adaptable and Reliable Systems

APPENDIX B. EXAMPLE DARTS TREES

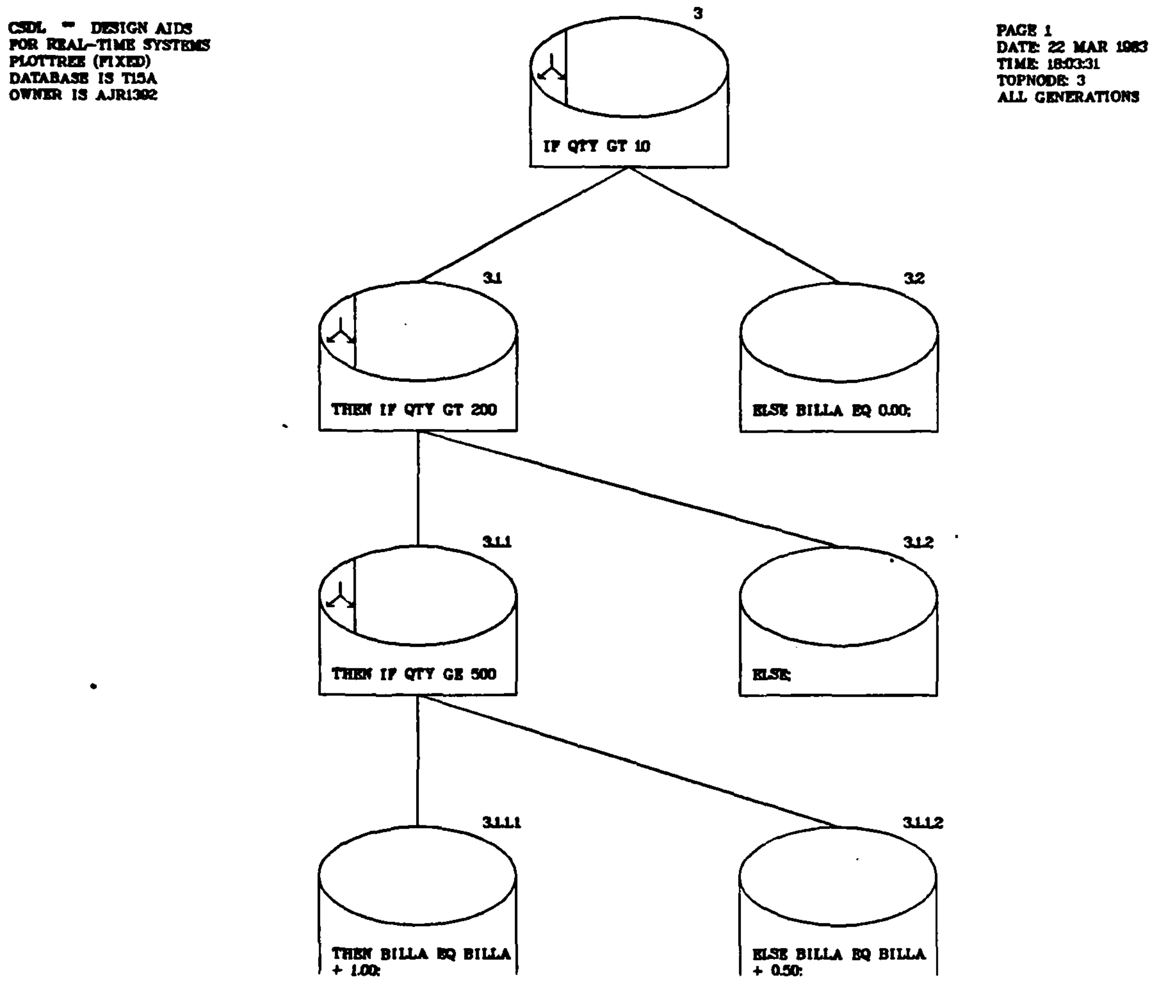


Figure 17. CACM Example 14a, DARTS Representation

```
CSDL ** DESIGN AIDS
FOR REAL-TIME SYSTEMS
PLOTTREE (FIXED)
DATABASE IS T15A
OWNER IS AJR1392
```

```
PAGE 1
DATE: 22 MAR 1983
TIME: 18:03:40
TOPNODE: 4
ALL GENERATIONS
```

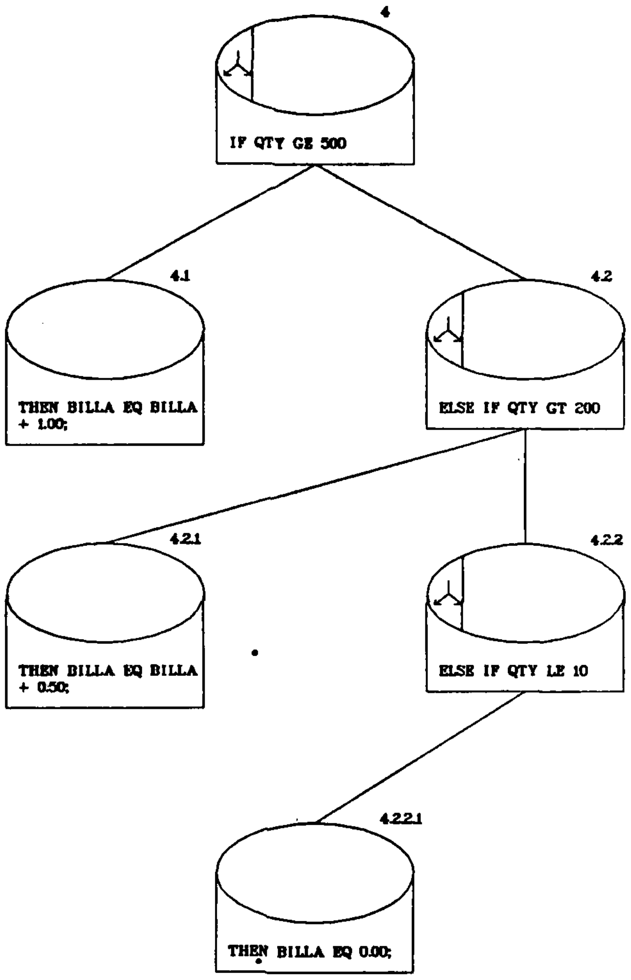


Figure 18. CACM Example 14b, DARTS Representation

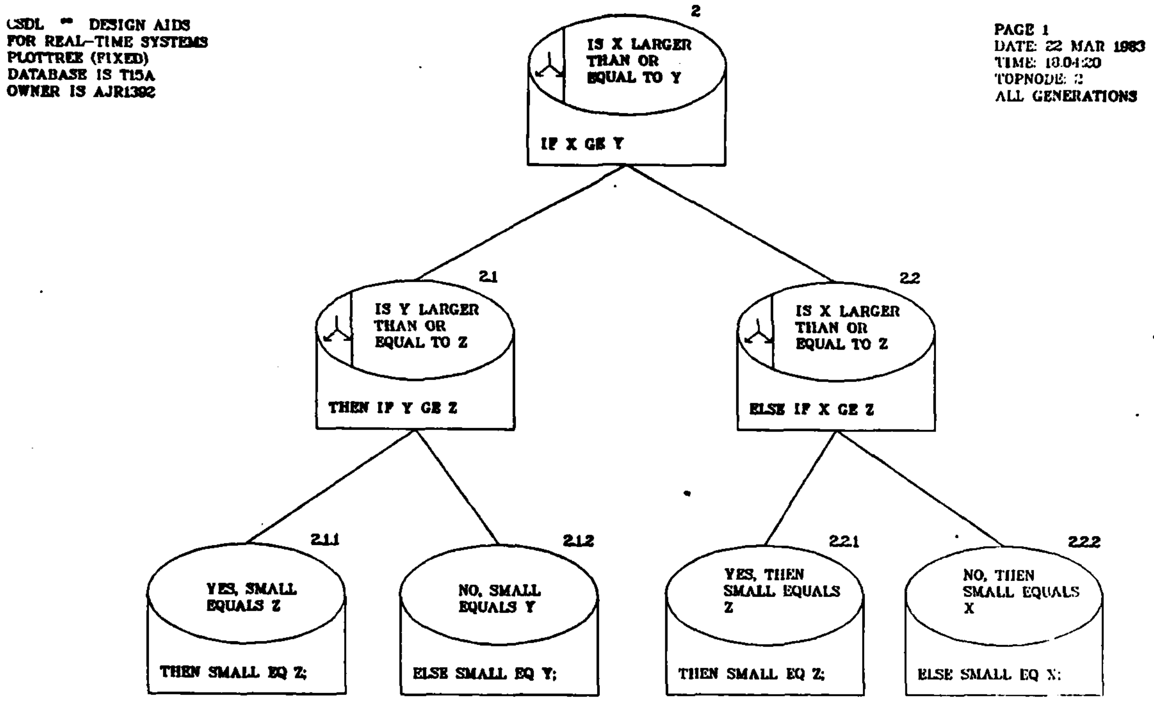


Figure 19. CACM Example 15a, DARTS Representation

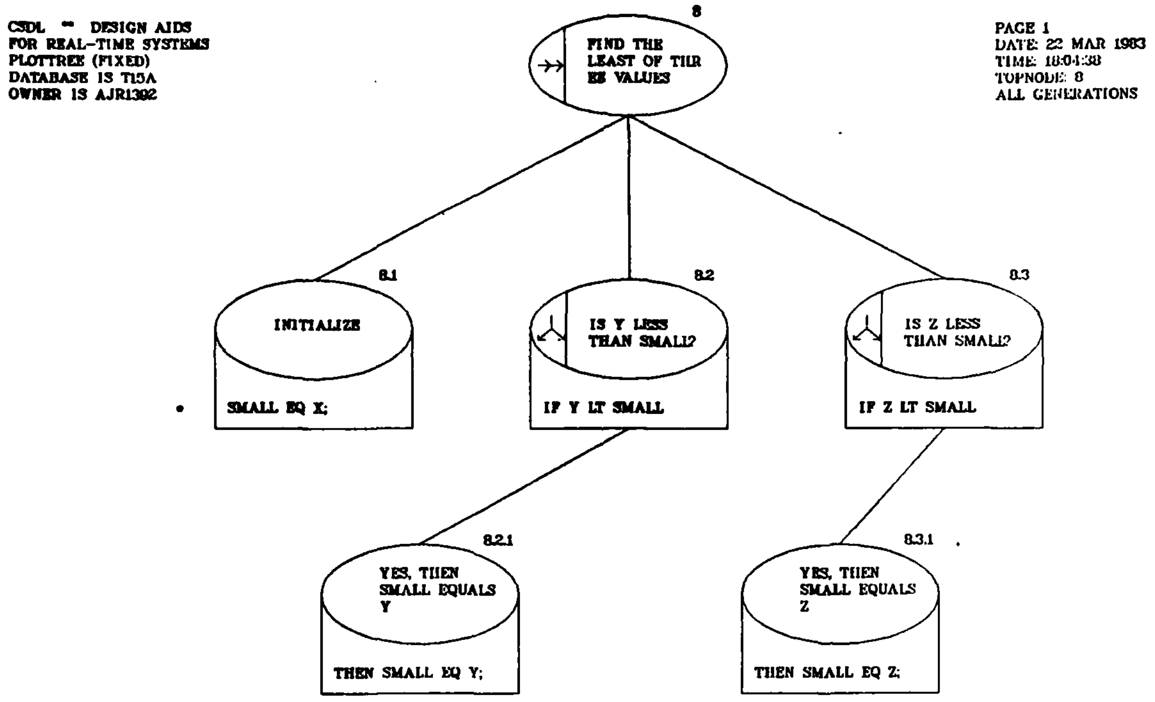


Figure 20. CACM Example 15b, DARTS Representation

APPENDIX C. DARTS PL/I HALSTEAD MODULES

C.1 DESIGN TREES

Figure 21 is the design tree for the implementation, in DARTS, of the Halstead metric. A description of the top four generations of nodes follows.

C.1.1 Node 9

Processing

The Halstead module determines the Halstead parameter values for a user-specified subtree of a design represented as a DARTS tree. The user also selects the counting method employed.

Input

Database representation of the tree.

User-specified top node of the subtree to be analyzed.

User-specified depth of the subtree to be analyzed.

User-specified counting method.

Output

On file FLOERR, the subtree designation and counting method, and the parameter values for the subtree.

C.1.2 Node 9.1

Processing

This node represents a recursive invocation of a tree traversal module. The tree traversal module visits each node of the user-specified subtree, collecting the numbers of operators and operands.

Input

Database representation of the tree.

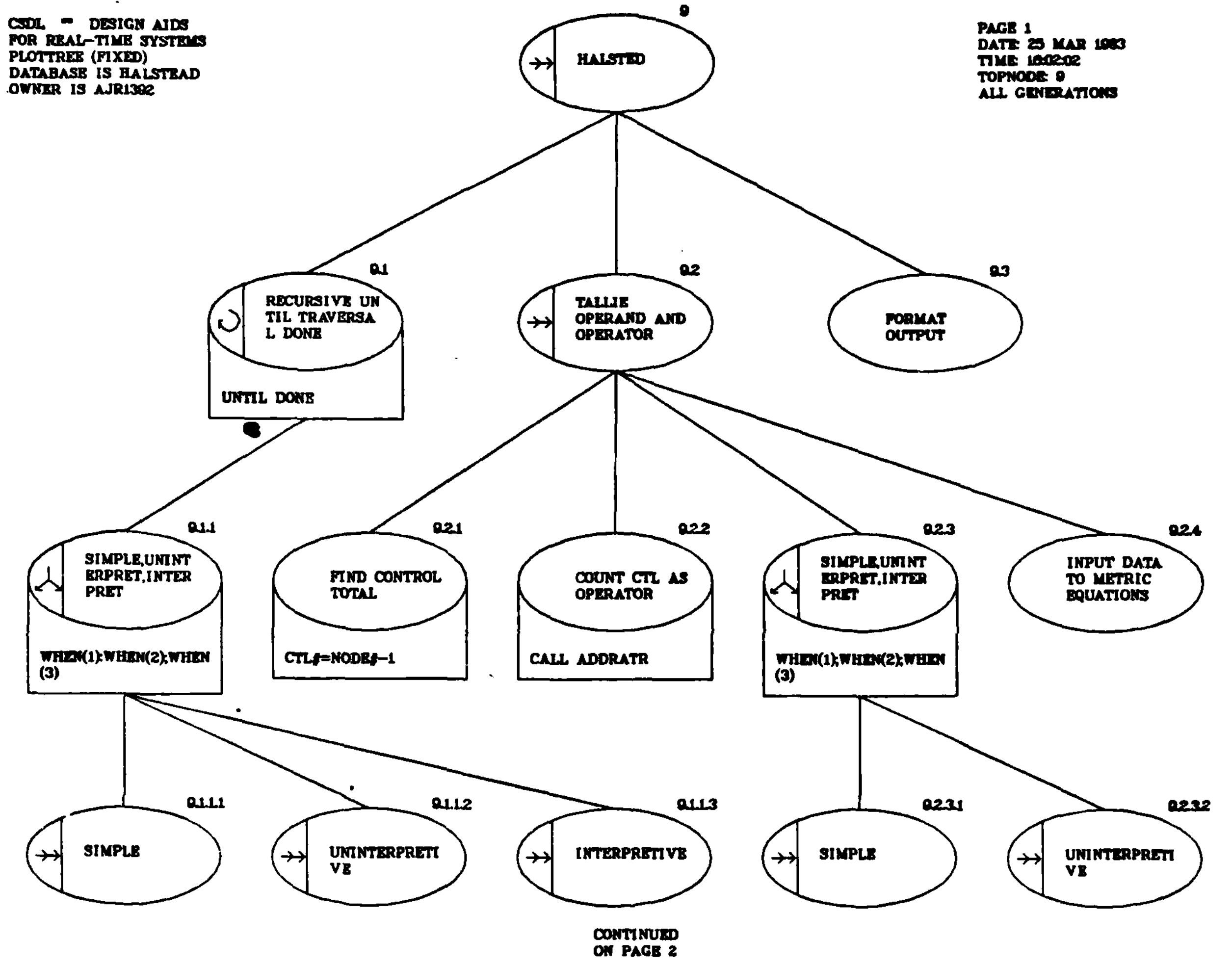


Figure 21. Design Tree for Halstead Metric. (Part 1 of 2)

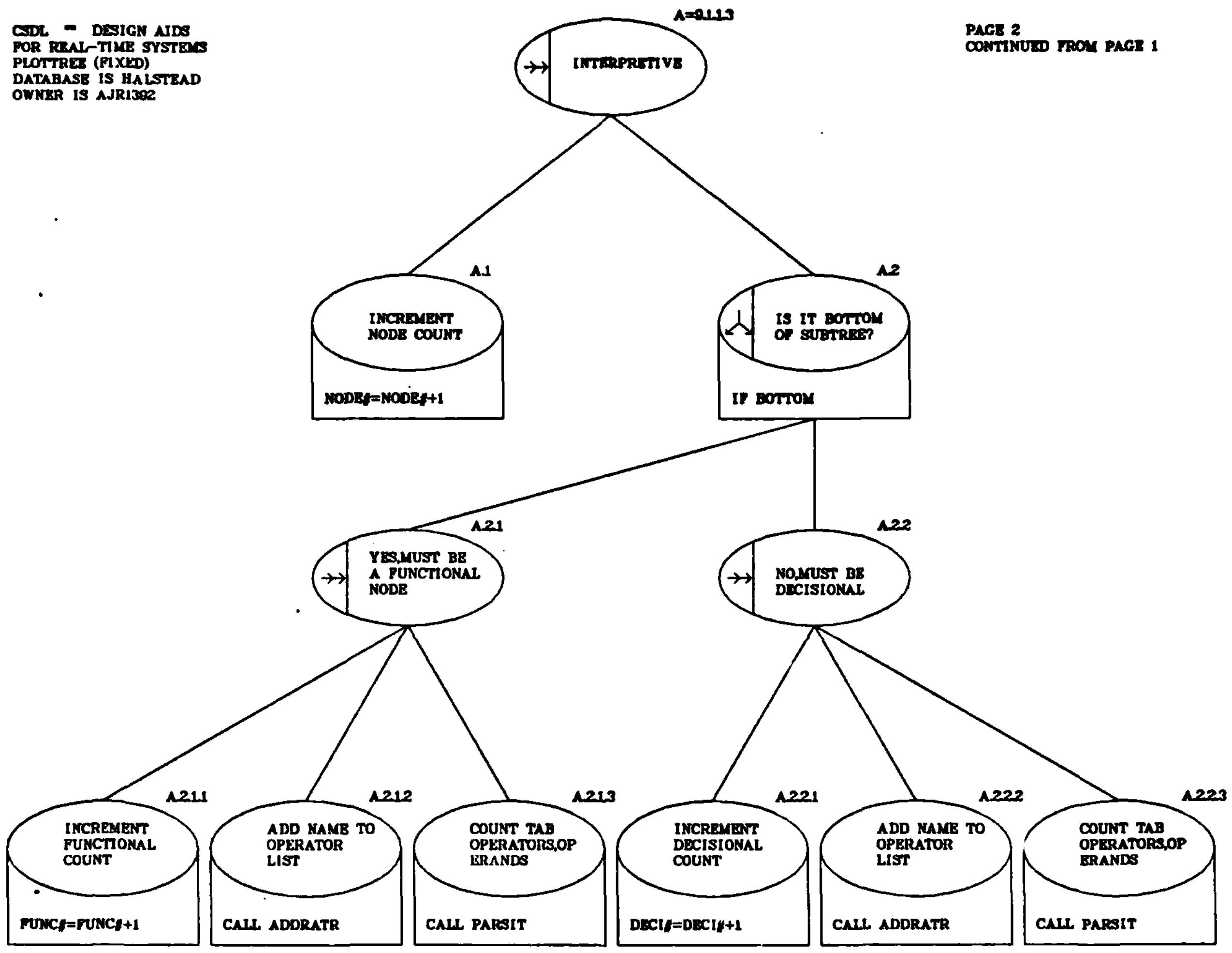


Figure 21. Design Tree for Halstead Metric. (Part 2 of 2)

Current top node of the subtree to be analyzed (initially the user-specified top node).
User-specified depth of the subtree to be analyzed.

User-specified counting method.

Output

List of operators with counts for each.

List of operands with counts for each.

C.1.3 Node 9.1.1

Processing

This node selects a counting routine to call depending on which counting method the user specified.

Input

User-specified counting method.

Output

C.1.4 Node 9.1.1.1

Processing

This module implements the simple counting method in which the variables in the INDATA and OUTDATA lists are counted as operands.

Input

Database representation of the tree,

List of operators with counts for each,

List of operands with counts for each.

Output

List of operators with counts for each,

List of operands with counts for each.

<u>C.1.5 Node 9.1.1.2</u>

Processing

This module implements the uninterpreted counting method in which nodes are differentiated as being function or decision nodes, but the node tabs are ignored. For function nodes, the variables in the INDATA and OUTDATA lists are counted as operands. For decision nodes, the variables in the PREDVAR lists are counted as operands.

Input

Database representation of the tree.

List of operators with counts for each.

List of operands with counts for each.

Output

List of operators with counts for each.

List of operands with counts for each.

<u>C.1.6 Node 9.1.1.3</u>

Processing

This module implements the interpreted counting method, based on the specification in Section 2.3.2.2, in which the node tabs are scanned for instances of operators and operands. INDATA, OUTDATA and PREDVAR lists are used to determine counts of operators and operands. The details of this processing are shown in Part 2 of Figure 21 on page 126

Input

Database representation of the tree.

List of operators with counts for each.

List of operands with counts for each.

Output

List of operators with counts for each.

List of operands with counts for each.

C.1.7 Node 9.2

Processing

This node adds the occurrences of flow-of-control to the operator count, and calculates the Halstead parameters from the basic operator and operand counts.

Input

List of operators with counts for each.

List of operands with counts for each.

Output

Halstead parameter values.

C.1.8 Node 9.2.1

Processing

This node calculates the number of flow-of-control instances from the number of nodes.

Input

Number of nodes traversed.

Output

Number of flow-of-control transfers.

C.1.9 Node 9.2.2

Processing

This node adds "CTL" to the list of operators.

Input

List of operators.

Output

List of operators.

C.1.10 Node 9.2.3

Processing

This node adds the number of flow-of-control instances to the number of operators accumulated for the subtree, depending on the counting method being used.

Input

Number of flow-of-control transfers.

List of operators with counts for each.

Output

List of operators with counts for each.

C.1.11 Node 9.2.3.1

See Node 9.1.1.1.

C.1.12 Node 9.2.3.2

See Node 9.1.1.2.

C.1.13 Node 9.2.4

Processing

This node calculates the Halstead parameters from the basic operator and operand counts.

Input

List of operators with counts for each.

List of operands with counts for each.

Output

Halstead parameter values.

C.1.14 Node 9.3

Processing

This node writes the Halstead parameters in tabular form onto the output file.

Input

Halstead parameter values.

Output

On file FLOERR, the subtree designation and counting method, and the parameter values for the subtree.

APPENDIX D. DARTS PL/I MCCABE MODULES

D.1 DESIGN TREES

Figure 22 shows four levels of the design tree for the implementation, in DARTS, of the McCabe metric. A description of each node follows.

D.1.1 Node 2

Processing

The McCabe subtree determines the McCabe metric interval bounds for a user-specified subtree of a design represented as a DARTS tree. The user-specified subtree is a module by definition, for this metric evaluation. A module info data area is created and stacked whenever a module is encountered in the user-specified subtree, as indicated by "BLK" or "SEG" in the node's tab. When the end of the module is encountered, an output line is created from the data in the module info data area, and the data area is popped off the stack and destroyed. Running totals of the numeric quantities are kept for the subtree being analyzed. They are printed at the end of the processing.

Input

Database representation of the tree.

User-specified top node of the subtree to be analyzed.

User-specified depth of the subtree to be analyzed.

Output

On file FLOERR, the module name, number of decisions, number of simple predicates and metric interval values for each module encountered, and the total metric interval value for the user-specified subtree. Modules which are invoked, but not present in the subtree, are tagged with a message instead of the metric interval value. This information shares file FLOERR with the Halstead metric output, but each function starts a new page when invoked.

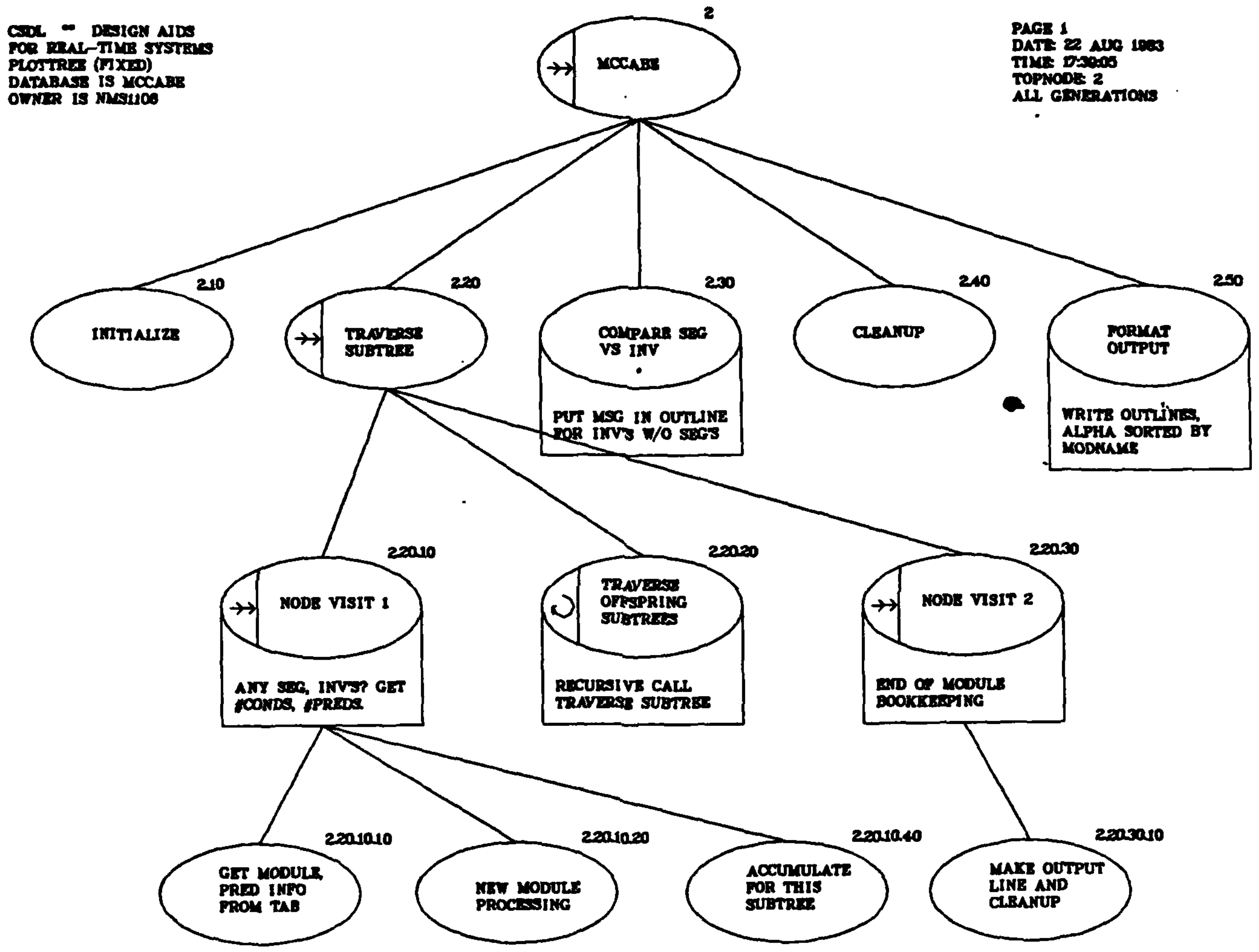


Figure 22. Design Tree for McCabe Metric.

D.1.2 Node 2.10

Processing

The Initialize subtree starts the output with a page header on a new page.

Input

None.

Output

Page header on file FLOERR.

D.1.3 Node 2.20

Processing

The Traverse subtree subtree visits each node of the user-specified subtree, collecting the metric data, and making the output lines. It also collects lists of the modules which occur in the subtree, and the modules which are referenced in the subtree.

Input

Database representation of the tree.

User-specified top node of the subtree to be analyzed.

User-specified depth of the subtree to be analyzed.

Output

Linked list of output lines for each module which occurs in the subtree, alphabetically ordered by the module name. Each output line contains the module name, the number of decisions, the number of simple predicates, and the metric interval value.

List of modules invoked in the subtree, in reverse of the order in which they are detected.

D.1.4 Node 2.20.10

Processing

The Node visit 1 subtree determines whether this node starts a module, whether it invokes a module, the number of decisions (possibly complex predicates) for the node, and the number of simple predicates for the node.

Input

Database representation of the tree.

Current module info data area.

List of modules invoked in the subtree.

Output

Current module info data area.

List of modules invoked in the subtree.

D.1.5 Node 2.20.10.10

Processing

The Get module pred info from tab subtree determines whether this node starts a module, whether it invokes a module, the number of decisions (possibly complex predicates) for the node, and the number of simple predicates for the node. It also adds items to the list of invoked modules, if this node invokes any. If there is a tab on the node, it gets the information from the tab. If there is no tab, the node does not start or invoke a module; the number of decisions is determined by the node type and number of offspring; and the number of predicates is the same as the number of decisions.

Input

Database representation of the tree.

Current module info data area.

List of modules invoked in the subtree.

Output

Current module info data area.

List of modules invoked in the subtree.

D.1.6 Node 2.20.10.20

Processing

The New module processing subtree creates and initializes a new module info data area if this node starts a module.

Input

Current (old) module info data area.

Output

Current (new) module info data area.

D.1.7 Node 2.20.10.40

Processing

The Accumulate for this subtree subtree adds the numbers of decisions and predicates for this node to the numbers of decisions and predicates accumulating for the current module.

Input

Current module info data area.

Output

Current module info data area.

D.1.8 Node 2.20.20

Processing

The Traverse offspring subtrees subtree recursively calls the traverse subtree module for each of the offspring of the current node.

Input

User-specified top node of the subtree to be analyzed.

User-specified depth of the subtree to be analyzed.

Output

Linked list of output lines for each module which occurs in the subtree, alphabetically ordered by the module name. Each output line contains the module name, the number of decisions, the number of simple predicates, and the metric interval value.

List of modules invoked in the subtree, in reverse of the order in which they are detected.

D.1.9 Node 2.20.30

Processing

The Node visit 2 subtree does any processing necessary at the return through the node. In this case, it turns the current module info data into an output line; destroys the current module info data area; and adds the numeric quantities for this subtree to those accumulating for the total line, if this node is the end of a module.

Input

Current module info data area.

Totals data area.

Output

Output line for the current module, containing the module name and metric interval value.

Totals data area.

D.1.10 Node 2.20.30.10

Processing

The Make output line and cleanup subtree turns the current module info data into an output line; destroys the current module info data area;

and adds the numeric quantities for this subtree to those accumulating for the total line, if this node is the end of a module.

Input

Current module info data area.

Totals data area.

Output

Output line for the current module, containing the module name and metric interval value.

Totals data area.

D.1.11 Node 2.30

Processing

The Compare SEG vs INV subtree creates an output line for every module which was invoked but not present in the subtree.

Input

List of output lines for modules present in the subtree.

List of modules invoked in the subtree.

Output

List of output lines for modules present in the subtree, with an output line added for each module which is invoked, but not present, in the subtree. The added lines contain the module name and a footnote reference number.

Flag indicating that the footnote message to that effect should be printed.

D.1.12 Node 2.40

Processing

The Cleanup subtree makes the total output line for the subtree analyzed.

Input

Totals data area.

Output

Totals output line.

D.1.13 Node 2.50

Processing

The Format output subtree writes the module and total output lines to the FLOERR file.

Input

Linked list of output lines for each module which occurs or is invoked in the subtree, alphabetically ordered by the module name.

Totals output line.

Footnote flag.

Output

On file FLOERR, the module name, number of decisions, number of simple predicates and metric interval values for each module encountered, and the total metric interval value for the user-specified subtree. If indicated by the flag, the "invoked but not present" footnote is printed.

APPENDIX E. DARTS DATA-FLOW TABLES FOR EXPERIMENT CONTROLLER EXAMPLE

Table 26. Data-Flow Table - Design 1 (Part 1 of 2)

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS DATA FLOW TABLE	TOPNODE ID: 1.1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 1 DATE: 08 SEPT 198 TIME: 11:21:39

TOPNODE: EXPERIMENT CONTROLLER 1	INDATA: EXP_TABLE EXP_COUNT $INT_B $INT_T $INT_U SENSOR_DATA	OUTDATA: $INTERVAL $STOP $STEP_CMD $REPORT $EXP_STATUS

NAME	INPUTS	PRODUCER	OUTPUTS	CONSUMER
INITIALIZATION			RESULTS_TABLE	MEASURE 1
GET CELL DATA 1	$SENSOR_DATA	SUPPLIED	SENSOR_DATA	MEASURE 1
MEASURE 1	RESULTS_TABLE SENSOR_DATA	INITIALIZATION GET CELL DATA 1	RESULTS_TABLE	MEASURE 2
CONTROL EACH EXPERIMENT	EXP_COUNT	MORE EXPERIMENTS		
INITIALIZE EXPERIMENT	EXP_TABLE	SUPPLIED	STEPS READINGS INTERVAL	POSITION BURETTE ENOUGH TAKE MEASUREMENTS TEST MEASURE START TIMER
POSITION BURETTE	STEPS	INITIALIZE EXPERIMENT ENOUGH		
SEND MOTOR COMMAND			$STEP_CMD	REQUESTED
WAIT FOR BURETTE INIT	$INT_B	SUPPLIED		
ENOUGH	STEPS	INITIALIZE EXPERIMENT ENOUGH	STEPS	POSITION BURETTE ENOUGH
START TIMER	INTERVAL	INITIALIZE EXPERIMENT	$INTERVAL	REQUESTED
TAKE MEASUREMENTS	READINGS	INITIALIZE EXPERIMENT TEST MEASURE		
WAIT FOR TIMER INIT	$INT_T	SUPPLIED		
GET CELL DATA 2	$SENSOR_DATA	SUPPLIED	SENSOR_DATA	MEASURE 2
MEASURE 2 (CONT)	RESULTS_TABLE	MEASURE 1	RESULTS_TABLE	MEASURE 2

Table 26. Data-Flow Table - Design 1 (Part 2 of 2)

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS DATA FLOW TABLE	TOPNODE ID: 1.1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 2 DATE: 08 SEPT 198 TIME: 11:21:39

NAME	INPUTS	PRODUCER	OUTPUTS	CONSUMER
(CONT)				
MEASURE 2	 SENSOR_DATA	MEASURE 2 GET CELL DATA 2	-	COMPUTE 1 LIST 1 COMPUTE 2 LIST 2
CHECK FOR USER INTERRUPT	$INT_U	SUPPLIED	USER_INTERRUPT	USER INTERRUPT PROCEDURE
USER INTERRUPT PROCEDURE	USER_INTERRUPT	CHECK FOR USER INTERRUPT		
COMPUTE 1	RESULTS_TABLE	MEASURE 2	STATISTICS	LIST 1
LIST 1	RESULTS_TABLE STATISTICS	MEASURE 2 COMPUTE 1	EXP_STATUS	SEND LIST TO PRINTER 1
SEND LIST TO PRINTER 1	EXP_STATUS	LIST 1	$EXP_STATUS	REQUESTED
CONTINUE				
TEST MEASURE	READINGS	INITIALIZE EXPERIMENT TEST MEASURE	READINGS	TAKE MEASUREMENTS TEST MEASURE
MORE EXPERIMENTS	EXP_COUNT	MORE EXPERIMENTS	EXP_COUNT	CONTROL EACH EXPERIMENT MORE EXPERIMENTS
STOP TIMER			$STOP	REQUESTED
COMPUTE 2	RESULTS_TABLE	MEASURE 2	STATISTICS	LIST 2
LIST 2	RESULTS_TABLE STATISTICS	MEASURE 2 COMPUTE 2	REPORT	SEND LIST TO PRINTER 2
SEND LIST TO PRINTER 2	REPORT	LIST 2	$REPORT	REQUESTED

Table 27. Data-Flow Table - Design 2 (Part 1 of 2)

```
CSDL  **  DESIGN-AIDS                      TOPNODE ID: 2.1                        PAGE 1
FOR REAL-TIME SYSTEMS                      ALL GENERATIONS                        DATE: 08 SEPT 198
DATA FLOW TABLE                            DATABASE IS:   SAMPLE                  TIME: 11:22:07
                                           USER IS: AJR1392
```

```
   TOPNODE: EXPERIMENT CONTROLLER 2          INDATA:  EXP_TABLE          OUTDATA: $INTERVAL
                                                      EXP_COUNT                   $STOP
                                                      $INT_B                      $STEP_CMD
                                                      $INT_T                      $REPORT
                                                      $INT_U                      $EXP_STATUS
                                                      $SENSOR_DATA
```

NAME	INPUTS	PRODUCER	OUTPUTS	CONSUMER
INITIALIZATION			RESULTS_TABLE	MEASURE COMPUTE 2 LIST 2
			FIRSTTIME	PREPARE FOR MEASUREMENT FIRSTTIME OR NEW EXP POST MEASURE PROCEDURE
			NEW_EXP	PREPARE FOR MEASUREMENT
			READINGS	TAKE MEASUREMENTS TEST MEASURE
CONTROL EACH EXPERIMENT	EXP_COUNT	MORE EXPERIMENTS		
PREPARE FOR MEASUREMENT	FIRSTTIME	INITIALIZATION SET FLAGS		
	NEW_EXP	INITIALIZATION INITIALIZE EXPERIMENT SET FLAGS NEW EXPERIMENT		
FIRSTTIME OR NEW EXP	FIRSTTIME	INITIALIZATION SET FLAGS		
CONTINUE				
INITIALIZE EXPERIMENT	EXP_TABLE	SUPPLIED	NEW_EXP	PREPARE FOR MEASUREMENT
			STEPS	POSITION BURETTE ENOUGH
			READINGS	TAKE MEASUREMENTS TEST MEASURE
			INTERVAL	START TIMER

Table 27. Data-Flow Table - Design 2 (Part 2 of 2)

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS DATA FLOW TABLE	TOPNODE ID: 2.1 ALL GENERATIONS DATABASE IS: SAMPLE USER IS: AJR1392	PAGE 2 DATE: 08 SEPT 198 TIME: 11:22·07

NAME	INPUTS	PRODUCER	OUTPUTS	CONSUMER
POSITION BURETTE	STEPS	INITIALIZE EXPERIMENT ENOUGH		
SEND MOTOR COMMAND			$STEP_CMD	REQUESTED
WAIT FOR BURETTE INT	$INT_B	SUPPLIED		
ENOUGH	STEPS	INITIALIZE EXPERIMENT ENOUGH	STEPS	POSITION BURETTE ENOUGH
START TIMER	INTERVAL	INITIALIZE EXPERIMENT	$INTERVAL	REQUESTED
WAIT FOR TIMER INIT 1	$INIT_T	SUPPLIED		
WAIT FOR TIMER INT 2	$INT_T	SUPPLIED		
TAKE MEASUREMENTS	READINGS	INITIALIZATION INITIALIZE EXPERIMENT TEST MEASURE		
GET CELL DATA	$SENSOR_DATA	SUPPLIED	SENSOR_DATA	MEASURE
MEASURE	RESULTS_TABLE SENSOR_DATA	INITIALIZATION MEASURE GET CELL DATA	RESULTS_TABLE	MEASURE COMPUTE 1 LIST 1 COMPUTE 2 LIST 2
POST MEASURE PROCEDURE	FIRSTTIME	INITIALIZATION SET FLAGS		
SET FLAGS			FIRSTTIME NEW_EXP	PREPARE FOR MEASUREMENT FIRSTTIME OR NEW EXP POST MEASURE PROCEDURE PREPARE FOR MEASUREMENT
CHECK FOR USER INTERRUPT	$INIT_U	SUPPLIED	USER_INTERRUPT	USER INTERRUPT PROCEDURE
USER INTERRUPT PROCEDURE	USER_INTERRUPT	CHECK FOR USER INTERRUPT		
COMPUTE 1	RESULTS_TABLE	MEASURE	STATISTICS	LIST 1
LIST 1	RESULTS_TABLE STATISTICS	MEASURE COMPUTE 1	EXP_STATUS	SEND LIST TO PRINTER 1
SEND LIST TO PRINTER 1	EXP_STATUS	LIST 1	$EXP_STATUS	REQUESTED

LIST OF REFERENCES

[Boe 81] Boehm, B. W., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J. 07632, 1981.

[Boe 83a] Boeing Aerospace Company, "Guidebook for Software Quality Measurement," Vol. II, February, 1983.

[Boe 83b] Boeing Aerospace Company, "Software Interoperability and Reusability," Vol. I, March, 1983.

[Boo 83] Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

[Cai 75] Caine, Stephen H., and E. Kent Gordon, 1975. "PDL--A Tool for Software Design," AFIPS Conference Proceedings Vol. 44, 1975 National Computer Conference.

[Cai 77] Caine, Farber, and Gordon, "PDL Program Design Language Reference Guide," Version 3, February, 1977.

[Cho 78] Chow, T.S., "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering Vol. SE-4, No. 3, May 1978, pp.105-109.

[Cou 83] Coulter, N.S., "Software Science and Cognitive Psychology" IEEE Transactions on Software Engineering Vol. SE-9, No. 2, March 1983, pp.166-171.

[CSDL82] The Charles Stark Draper Laboratory, Inc., 12 January 1982. Design Aids .for Real-Time Systems (DARTS): A User's Guide, Version 3. CSDL-C-5441, Cambridge, MA. January, 1982.

[DoD 82a] Department of Defense, "Strategy for a Software Initiative," 1 October, 1982.

[DoD 82] MIL-STD-SDS Joint Logistics Commanders, Proposed Military Standard on Defense System Software Development, and attached Data Item Descriptions, 15 April, 1982.
R-DID-107 Software Requirements Document
R-DID-110 Software Top Level Design Document
R-DID-111 Software Detailed Design Document

[DoD 83] Department of Defense, "Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy," 1 April, 1983.

[Els 78] Elshoff, J. L., "An Investigation into the Effects of the Counting Method Used on Software Science Measurements," SIGPLAN Notices, Vol. 13, No. 2, pp. 30-45, February, 1978.

[Fit 78] Fitzsimmons, A., and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys, Vol. 10, No.1, pp. 3-18, March, 1978.

[Fur 81] Furtek, F. C., J.B. DeWolf, and P. Buchan, "DARTS: A Tool for Specification and Simulation of Real-Time Systems," Proceedings of the AIAA Computers in Aerospace III Conference, October, 1981.

[Gaf 81] Gaffney, J. E., Jr., "Software Metrics: A Key to Improved Software Development Management," Computer Science and Statistics, Proc. 13th Symposium on the Interface 1981, Springer-Verlag, New York, pp. 211-220.

[Gor 83] Gordon, M., "The Byron Program Development Language," Journal of Pascal and Ada, pp. 24-28, May/June 1983.

[Gor 79] Gordon, R. D., "Measuring Improvements in Program Clarity", IEEE Trans. Software Engineering, Vol. SE-5, pp. 79-90, March, 1979.

[Hal 77] Halstead, M. H., Elements of Software Science, Elsevier North Holland, Inc., Operating and Programming Systems Series, New York, 1977.

[Hal77a] Halstead, M. H., "A Quantitative Connection Between Computer Programs and Technical Prose," Proceedings of Fall COMPCON 1977, pp. 332-335.

[Hal 78] Halstead, M. H. "Management Prediction - Can Software Science Help?" Proceedings IEEE COMPSAC 78 (2nd International Computer Software and Applications Conference), 1978, pp 126-128.

[Hal 78a] Halstead, M. H. "Software Science -- A Progress Report," Second Software Life Cycle Management Workshop, Atlanta, GA, August, 1978, pp. 174-179.

[Ham 82] Hamer, P.G. and G.D. Frewin, "M.H. Halstead's Software Science - A Critical Examination," Proc. of the Sixth International Conference on Software Engineering,, Tokyo, Japan, pp. 197-205, September 1982.

[Ker 74] Kernighan, B. W., and P.J. Plauger; "Programming Style: Examples and Counterexamples," ACM Computing Surveys, Vol. 6, pp. 303-319, December, 1974.

[McC 76] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, pp 308-320, December, 1976.

[McC 80] McCall, J. A., and M. Masumoto, Software Quality Metrics Enhancements, RADC-TR-80-109, Rome Air Development Center, 1980.

[McC 77] McCall, T. J., P. K. Richards and G. F. Walters, Factors in Software Quality, GE Technical Information Series 77CIS02, June, 1977.

[McC 79] McCall, T. J., and M. T. Matsumoto, Software Quality Metrics Enhancements Final Report, September, 1979.

[Men 75] Mendelbaum, H.G., and F. Madaule, "Automata as Structured Tools for Real-Time Programming," Proc. of the 1975 IFAC-IFIP Workshop on Real-Time Programming , Pittsburg, PA. August, 1975, pp. 59-65.

[Mye 77] Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, The Association for Computing Machinery, Inc., pp 61-64, October, 1977.

[San 83] San Antonio, R. C., and K. L. Jackson, "Application of Software Metrics During Early Program Phases," Proc. of NSIA OSD Conference, Washington, D. C., February, 1983.

[She 83] Shen, V.Y., S.D. Conte, and H.E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," IEEE Transactions on Software Engineering Vol. SE-9, No. 2, March 1983, pp.155-165.

[Szu 80] Szulewski, P. A., M.H. Whitworth, P. Buchan, and J.B. Dewolf, Quality Assurance Guidelines and Quality Metrics for Embedded Real-Time Software Designs, CSDL-R-1376, The Charles Stark Draper Laboratory, Inc., May, 1980.

[Szu 81] Szulewski, P. A., M.H. Whitworth, P. Buchan,and J.B. Dewolf, "The Measurement of Software Science Parameters in Software Designs," ACM SIGMETRICS Performance Evaluation Review, Vol. 10, No. 1, Spring, 1981.

[Tau 77] Tausworthe, Robert C., Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Tei 77] Teichrow, D., and E. Hershey, "PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information Processes Systems," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp 41-48, January, 1977.

[Weg 82] Wegner, P., "Ada Education and Technology Transfer Activities," ACM Ada Letters, September, 1982.

BIBLIOGRAPHY

(1) Barnes, J. G. P., Programming in Ada, Addison-Wesley, 1982.

(2) Beser, Nicholas, "Foundations and Experiments in Software Science", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 48-72.

(3) U. S. Department of Defense, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 22 January 1983, American National Standards Institute, Inc., New York NY 10018.

(4) Downes, V. A. and S.J. Goldsack, Programming Embedded Systems with Ada, Prentice/Hall, 1982.

(5) Gilb, T., Software Metrics, Winthrop Publishers, Inc., Computer Systems Series, Cambridge, MA., 1976.

(6) Gross, D. R., M. A. King, M. R. Murr and M. R. Eddy, "Complexity Measurement of Electronic Switching System (ESS) Software", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 75-85.

(7) Hartman, S. D., "A Counting Tool for RPG", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 86-100.

(8) Howden, W. E., "Contemporary Software Development Environments", Communications of the ACM, Vol. 25, No. 5, May 1982 pp 318-329.

(9) IIT Research Institute, "Software Engineering Research Review: Quantitative Software Models,", order no. SRR-1, Data & Analysis Center for Software (DACS), Rome Air Development Center, Griffiss AFB, New York, 1979.

(10) Intermetrics Incorporated, Byron Reference Manual, Cambridge, MA 02138, August, 1983.

(11) Laurmaa, Timo and Markku Syrjanen, "APL and Halstead's Theory: a Measuring Tool and Some Experiments", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 32-47.

(12) Naib, F. A., "An Application of Software Science to the Quantitative Measurement of Code Quality", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 101-128.

(13) Schnurer, K. E., "Product Assurance Program Analyzer (P.A.P.A.) A Tool for Program Complexity Evaluation (abstract only)", ACM SIGMETRICS Performance Evaluation Review, Vol. 11, No. 3, Fall 1982, pp 73-74.

(14) Yourdon, Edward and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.